UNIVERSITY OF OSLO
Department of Informatics

Migrating Video Streaming Sessions in Planning-Based Middleware

Master thesis

Håkon S. Ulvestad

2nd of May 2008

Abstract

More and more people are watching media content over the internet or from own personal media servers, and this way is slowly taking over for traditional television broadcasts. With this, the desire and possibility to watch media content from any location at any time arises as people are often on the move. With todays technology everything from computers to television sets and even hand-held devices have access to the internet and capability for decoding and rendering video. The problem addressed in this master thesis arises when the user decides from one location to another location, for example from the living room to the bedroom. The user might want to continue to watch what he was already watching in the living room when he gets to the bedroom.

This problem gives root to another desired service; being able to move a media streaming session from one device, for example a television set, to another, for example a hand held PDA. This is often known as client side session migration or session hand-off. Ideally, such functionality should work with as little user interaction and as seamlessly as possible.

This thesis proposes a way of adding session migration functionality to an existing media streaming application, the so-called Personal Media Service (PMS). The PMS application uses the QuA middleware platform to automatically adapt media quality to the client's context. It is proposed that each device can be represented by a QuA service mirror. In this thesis the QuA functionality is expanded to also use the user's and devices' physical locations to decide what device the media server is to stream to.

As is shown through the design, implementation and testing done in this thesis, using planning based middleware for session migration by representing each device as a service mirror will work. Bandwidth, and user and device positions, allow the planning-based middleware to automatically determine which device will give the best user experience at any time. The tests show that performance of session migration using the QuA middleware does not scale very well for many available devices, but works sufficiently well for few devices.

Acknowledgements

There are several people who have helped make this thesis possible and deserve thanks.

First of all I would like thank my supervisors professor Frank Eliassen and post doctorate Viktor S. Wold Eide for their invaluable support and insight during the work with the master thesis. Without their extensive knowledge and expertise, this theses would not have been possible.

I would also like to thank PhD student Eli GjÃ¸rven who has extensive knowledge of the QuA implementation, and has helped me with issues and problems encountered when working with the QuA middleware.

Thanks are also in order to master student Kristoffer Furuheim who took time out of his busy schedule, right before delivering his own master thesis, to read through my thesis and provide feedback on language and grammatical issues.

I would also like to thank Romerike Taekwondo for providing me with something else to do, and taking my mind away from the stressful work on this thesis a few times each week.

Ulvestad Konfeksjon, my employer, also deserve thanks for letting me focus on the master thesis even though there was very much work to be done.

Finally I would like to thank my family and friends for supporting me through this time. They are the backbone in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

More and more people are watching digital media content over the internet or from own personal media servers, and this way is slowly taking over for traditional television broadcasts. With this, the desire and possibility to watch media content from any location at any time arises as people are often on the move. With the technology of today, everything from computers to television sets and even hand-held devices have access to the internet and capability for decoding and rendering video.

People generally tend to want to watch their media content from the couch in the living room rather than sitting in front of their computer screens, and streaming media from a personal media server to for instance a television set is becoming common. The video stream could as easily be a movie transferred from a file on the local network as a live news broadcast streamed over the internet.

The problem addressed in this master thesis arises when the user decides to move to another location, e.g. from the living room to the bedroom. The user might want to continue to watch what he was already watching in the living room when he gets to the bedroom. If this was a live video broadcast received over the internet, setup would be required to receive the video on the new device (the television set in the bedroom).

This scenario gives root to another desired service; being able to move a media streaming session from one device, e.g. a television set, to another, i.e. a hand held PDA. This is often known as client side session migration or session hand-off. This functionality should work with as little user interaction and as seamlessly as possible. Figure 1.1 illustrates an environment where session migration might be desirable. Video is provided by a content provider and the user has several options to where he could receive

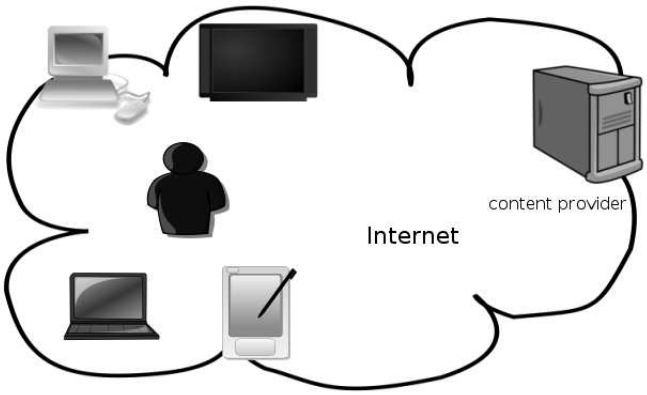

Figure 1.1: Environment where session migration might be desirable

the video, depending on context the user might want the receiving device to change during a video streaming session.

The QoS-Aware Component Architecture (QuA) platform, developed by Simula Research Laboratory, is a planning-based reflective middleware. It separates the concerns of developing the application logic from handling the applications resource demands (QoS requirements). This makes developing context aware and evolving systems easier as QuA handles adaptation and reconfiguration of the implemented services of the application.

PMS (Personal Media Service) is an application developed to illustrate of how the QuA platform technology can be used for applications with adaptation needs. PMS is a media server application streaming video data to a user over a network. The video data streamed by the PMS server is adapted according to the current available bandwidth using a concept known as scalable video coding (SVC). In common media streaming systems the user often has to pause the video to buffer the incoming video before rendering it if the bandwidth is too low for live streaming of the video. This is the issue that the PMS solution addresses. By using SVC, PMS can continue streaming and rendering video without buffering, even if the bandwidth is too low to handle the full stream. This is done by scaling the video and streaming it with a lower quality reducing the demands for bandwidth. By using QuA, PMS will adapt the video to stream the highest possible quality of video allowed at all times.

This thesis proposes a way of doing session migration for the existing PMS application. The PMS application uses the QuA middleware platform to

adapt media quality to the client's context. In this thesis the QuA functionality will be expanded to also use the user's and devices' physical locations to decide what device the media server is to stream to.

The rest of this chapter is structured as follows; Section 1.1 describes the objectives and goals of this master thesis and section 1.2 describes how these goals are to be achieved. Section 1.3 defines the scope and limitations of this thesis. Section 1.4 provides a summary of the results of this thesis. Finally section 1.5 gives a guide to the structure of the rest of this thesis document.

1.1 Problem statement

The focus of this thesis is to investigate the feasibility of using planning-based middleware to perform session migration on sessions in a personal media streaming system. The main problem statement is as follows:

How can session migration be implemented for video streaming services using planning-based middleware?

To find the answer to this question, some things will have to be researched. This section presents some questions that needs to be answered.

- How can devices be represented in the planning-base middleware?
- What context is essential to making the correct decisions, giving the best user experience?
- How can session migration be done in as seamless a manner as possible without losing any data?

To give an answer to the last question, research and discussion different possible solutions is required.

Accordingly, the hypothesis to be tested in this thesis becomes as follows:

Session migration can be implemented using planning-based middleware by representing devices as service mirrors, and using bandwidth, and user and device positions, for planning.

1.2 Research method outline

The main method used to solve the problem which is the focus of this master thesis, is to implement a working solution for session migration for the PMS

application and evaluate its performance.

The first step for doing this is a study phase where thorough investigation into the subjects which lay the basis for session migration and the PMS application are done. To fully understand how the PMS application works, extensive research into PMS, component architecture middleware, mainly the QuA middleware, is necessary. This will be done by studying research articles which cover previous work in these areas. Once a basic understanding of the basis for the PMS application is reached, the existing code for QuA and PMS is studied. Both knowledge of the implementations of QuA and PMS, as well as the underlying theory, is important for realizing the final implementation which is the focus of this thesis.

Following the study phase is the design phase, where a the session migration functionality is discussed, and a design and model is outlined. This design is the basis for doing the final implementation.

After the design phase, the design presented is implemented. Testing of the implementation is done for each component as it is implemented. During this phase issues with the design are discovered and changes to the design might be needed. This might make further research and study work necessary. Changes are done to the design and implemented.

When the implementation is done, a testing phase will follow. In this phase tests are designed to see the implementation's performance and behavior according to the requirement specification. During this phase, any problems with the implementation are discovered and will have to be corrected.

The final phase is the evaluation phase where the results of the test are discussed. Solutions for unexpected or inferior performance revealed in the tests are discussed and proposed. The results of the tests in accordance with the requirement specification are the basis for the conclusion of this master thesis. As there is no suitable other implementation for session migration to test against, this will be left for future work.

1.3 Limitations/scope

Due to constraints on how much time can be used for completing a master thesis, some limitations as to which areas will be covered in this thesis has been set. This section explicitly defines the scope of this master thesis as to which parts will be researched and considered for the final implementation of session migration.

1.3.1 Session migration

Session migration is sometimes also referred to as session transfer or session hand-over. The term *session migration* will be used in this thesis. The form of session migration covered in this master thesis is the migration of a video streaming session from one device to another. Session migration involving changing networks, as well as session migration involving migrating a stream from one server to another, will not be considered in this master thesis. However, the background chapter will give brief introductions to these subjects.

1.3.2 Middleware

This master thesis will strictly focus on the QuA middleware. Other middleware alternatives will not be considered or researched. The reason for this is that the implementation part of this thesis, is expanding the already existing PMS application to also support session migration. As the PMS application already exists and is using the QuA middleware (in fact it was created to test the architecture) there really is no other choice for middleware. This master thesis will further test the middleware's capabilities and might discover changes that needs to be done to the middleware.

A brief introduction and explanation of the concept of component architecture middleware will be done in the background chapter.

1.3.3 Session splitting and duplication

Video is very often accompanied by audio. Performing session migration for a video stream that contains both audio and video would certainly be more complex than for just video. As the current PMS application does not support audio, this will not be implemented or designed for. However, the subject will be touched briefly, as audio is very important to video streaming and it gives additional advantages to the use of session migration. One such advantage would be the possibility to split a stream, streaming the video data to the television and the audio data to the home cinema surround system, giving the user an enhanced experience compared to playing both the audio and the video on the television set.

There are also situations where splitting a streaming session might be desirable. For instance, one might want to duplicate a session and have the same video data sent to two separate devices. Section 3.1 gives one scenario for this. Having the stream duplicated could also be effective to achieve smooth

handover, which will be described in the design chapter of this thesis.

Splitting a session in two, or more, introduces some seriously complex issues. For instance duplicating a session into two equal sessions, would double the server's out band bandwidth needs. If the available bandwidth is less than this, this would force the server to send the video at a lower quality, which is not desirable. It would also induce heavier complexity to the implementation especially when it comes to the way this should be handled by QuA.

Due to these complexities, splitting and duplication of sessions will not be considered for the final implementation. However, it will be discussed briefly, as these functionalities might be highly desirable.

1.3.4 Security

Security is a very important subject today when it comes to distributed systems and applications. Security implications on the PMS application would for instance be authorizing clients when connecting to the server, and preventing malicious users from manipulating the server to hijack the stream or gain access to the other devices available to the server. There are loads of other ways that such a system might be exploited. However, this is a much to broad subject to be considered within the scope of this master thesis. In the case that the PMS application should ever be commercialized and put to real use, these are concerns that should definitely be handled.

1.3.5 Signaling

Communication and signaling mechanisms will not be researched to great extent in this master thesis, though one approach using SIP will be presented in the background chapter. The focus of this thesis is session migration and planning for PMS and due to time limitations only a simple, but working, mechanism for communication between the server and the client application will be implemented.

1.4 Summary of results

Through design, implementation and testing session migration for PMS, the hypothesis presented in this chapter is shown to be true. When using planning based middleware for session migration, representing each device as a service mirror will work. Bandwidth, and user and device positions, is shown

to give a good indication of what device that will give the best user experience. There are, however, other context that one might be introduce to give even better results.

The performance of session migration using the QuA middleware does not scale very well for many available devices, but works sufficiently well for few devices. There are some changes/additions that could be done to QuA to improve performance.

1.5 Thesis structural overview

This thesis is structured as follows:

Chapter 2 presents the background material necessary to understand the work of this thesis.

Chapter 3 presents the requirement specifications set for the final implementation of session migration in PMS.

Chapter 4 presents and discussed the design of the session migration functionality.

Chapter 5 gives a walkthrough and discussion of the implementation done for this thesis; including issues discovered during the implementation phase as well as changes made to the existing application.

Chapter 6 presents the testing, and results of these, done to the implementation. These tests focus on correctness, time taken for session migration and scalability.

Chapter 7 discusses and evaluates the design and implementation of session migration based on the results of the tests in chapter 6.

Chapter 8 concludes the thesis and suggests further work.

Appendix A contains a CD with the source code for the PMS as well as the QuA code which is needed to run the PMS application.

Chapter 2

Background

This section contains a summary of previous research work done in the fields that are most relevant to the research done in this thesis. The research fields that will be presented in this chapter are: QuA, PMS, Session migration, and Smart Homes.

2.1 QoS-aware Component Architecture (QuA)

To use QuA to implement dynamic session migration based on context it is important to know how QuA and its components are functioning. Knowing how service mirrors can be used to represent different devices and how QuA selects the appropriate one based on the current context are vital to be able to implement a working session migration functionality for PMS which runs on the QuA platform. In this section we will study the most important aspects of QuA needed for this.

2.1.1 Overview

QuA (QoS-Aware Component Architecture) is a planning-based middleware, developed at the Simula Research Laboratory, using component architecture to handle Quality of Service (QoS) sensitive applications. It separates the concerns of developing the application logic from handling the applications resource demands (QoS requirements). This means that the developer should need no knowledge of the underlying architecture for QoS (sensing context variables etc.), but still needs to create components for doing the applications specific handling of different levels of quality on the available resources. This is done by creating a set of plans (mirrors) for different qualities of resources

for a given service [3].

An example of this in PMS could be to stream video at full quality when bandwidth is high, and reduce the quality of the video (by for instance reducing luminance or chromance quality of the video) when the bandwidth drops beneath a certain level. These demands need to be set up by the application developers. It is the QuA-platform's task to select which of these plans are the best fit (yield the highest utility) for the current context. A utility function calculates a value measuring how good a service is performing according to its specification. A utility of 1 would mean that the video streamer is streaming the video at highest quality without any delay. A utility of 0 will mean that the service is not performing at all, no video is being streamed.

QuA defines services as something which takes some input and provides an output. A service can range from anything like simple addition of numbers to encoding of video. In QuA services are implemented as components. *Components* can be composed together creating a *composition*. Components defines the interfaces and dependencies of a service. Such a dependency could for instance be the need for a Java runtime environment for a service that is implemented as a Java component.

QuA was developed to support evolution of systems. In an evolving system a new version of an already running service is advertised. The motivation behind QuA is allowing the middleware platform to evaluate and use this new service version during runtime allowing a system to evolve without shutdown or altering of the rest of the running system. Evolution of systems can be divided in *substitutional* and *non-substitutional* evolution. If the type of a new service conforms to the type of the old service the evolution is known as substitutional. If the type does not conform to the old type it is known as non-substitutional evolution [3]. QuA supports both types of system evolution.

QuA uses pluggable core services to allow QuA support for different platforms. This means that different implementations of each of the core components of QuA like the *Implementation Broker* and the *Service Planner* can be "plugged" into the QuA core. This allows QuA to run on different platforms with different dependencies. QuA allows distributed services by allowing core QuA funtionality on different hosts. For example, Johannes Oudenstad implemented a QuA Implementation Broker using peer-2-peer technology, allowing an instance of QuA to obtain *service mirrors* for a given service from remote hosts, in his Master thesis [14].

A brief walkthrough of the most important components of the QuA architecture follows.

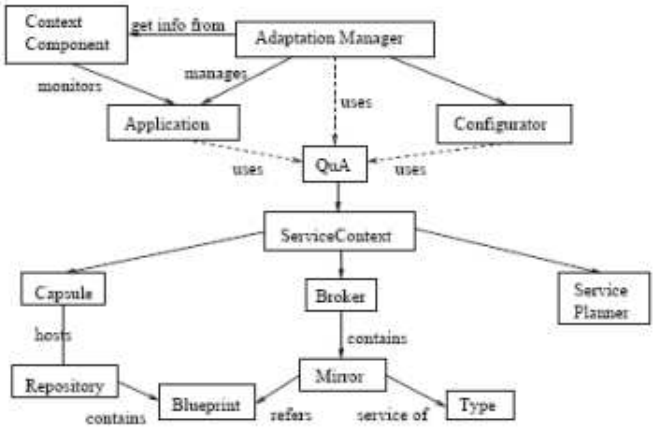

Figure 2.1: The QuA middleware. Figure taken from [3].

2.1.2 Service Context

The service context QuA component provides access to the execution environment of a running service on the QuA platform. As seen in figure 2.1 the service context contains a QuA *capsule*, an *implementation broker* and a *service planner*. Each of these components will be explained in detail in following sections.

2.1.3 Capsule

A *capsule* is a QuA component which provides a runtime environment for instances of the QuA platform. A capsule has one or more repositories which store the *blueprints* (described in section 2.1.5) for services. Each instance of a capsule might have different capabilities. For instance if a service is implemented in Java, a capsule that provides a Java runtime environment will be needed to successfully deploy and run it. Currently only a Java, and a Smalltalk, platform has been implemented for QuA, so all components created to work with the QuA middleware needs to be coded in Java or Smalltalk, unless one implements a capsule to allow other programming languages.

The capsules can be advertised to the service planner as service mirrors. This will allow the service planner to find the correct capsule to host a service mirror to resolve all the requirements of the service being planned for.

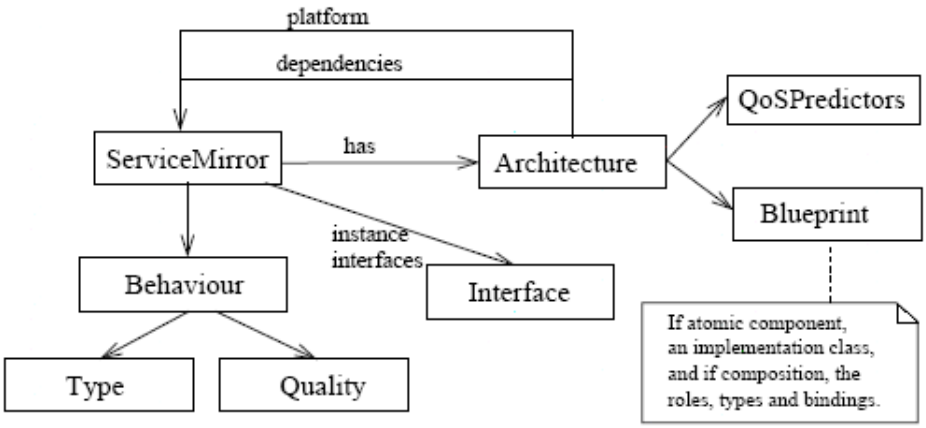

Figure 2.2: QuA Service Mirror. Figure taken from [3].

2.1.4 Service mirror

Service mirrors provide mirror-based reflection on services in the QuA platform. Several service mirrors may be advertised to the broker for each service type and each represents a different implementation of the specified service (specified by the service type). Figure 2.2 shows the architecture of a QuA service mirror. A service mirror has three parts; the *architecture* part, the *interface* part and the *behaviour* part.

The *behaviour* part represents the type of the service, which includes the functional behaviour and the non-functional behaviour of the service. The functional part specifies the behaviour of the mirror according the the interfaces given in the interface part. The non-functional behaviour represents the qualitative behaviour of a service. This is done in terms of QoS dimensions and a utility function. Different QoS dimensions and preferences can be set for each implementation of a type.

The *architecture* part specifies the component blueprint, dependencies and the capsule type. The dependencies contains information about what other services (service mirrors) that the specific service requires to function. The capsule dependency, which is also represented as a service mirror, is important for letting the service planner know the runtime needs of the service.

The *interface* part has reference to an object that provides the interface specified by the service type. The interface is needed to know what functionality a service has, and what information is needed to use them, such as functions

in a Java object with specified input and output variables.

Service mirrors in QuA can be in five different states; *specified*, *architected*, *provisioned*, *assembled*, and *running state*. Only a service mirror which is in the *running* or *provisioned* state will be considered by the *service* planner during planning. A complete explanation of these states can be found in [7].

2.1.5 Blueprint

A *blueprint* is a a binary representation of a component, interpreted by the component platform. A blueprint may contain code, references to code and/or specifications. A blueprint may also represent a composition of components, specifying each of the component services and their bindings. A blueprint is instantiated in the capsule which provides a running instance of the platform, meeting all the requirements of the *architecture* part of the service mirror.

Blueprints often contain pre-compiled code for a service component which is loaded from a binary file. Since only a Java capsule is currently implemented, only Java components are supported. Thus, a blueprint for a component is a byte string containing the jar file (Java archive) made when compiling a component's code. As described earlier the interface, as well as the capsule, of these components must be explicitly declared in the service mirror for QuA to know how to correctly interpret and execute the code contained in the blueprint.

2.1.6 Implementation broker

All service mirrors are managed by an implementation broker. When a new service mirror is discovered/created it is advertised to the implementation broker which adds it to the brokers collection of mirrors. As the QuA middleware is not fully implemented some desired functionality is missing. One such functionality is removing service mirrors from the broker.

The implementation broker provides functionality like *getMirrorsFor(ServiceMirror)* which returns all service mirrors that conform to the service type of the given service mirror. This is used by the service planner to retrieve all possible alternative implementations for the service it performs planning for.

2.1.7 Context componentes

Context sensing components are essential to the QuA middleware since they provide the context information which is used for planning and adapting an application. Context in QuA could be anything; available bandwidth, light and temperature conditions, physical location etc. Each of these context need specialized sensing components.

The QuA platform supports pluggable core components which allows for easy implementation of any given context sensing component. Context sensing components could easily be dummy components which only simulate context and context change for the development phase of the application and then be substituted by components that measure actual context.

The QuA platform maintains a context repository, which is currently realized as a simple *Map* structure. The context sensing components dynamically update this context repository as it measures context. The context repository is used by other QuA components, such as the adaptation manager and utility functions, to retrieve context information.

2.1.8 Service Planner

The *service planner* is, as the name implies, in charge of planning the service. That is, evaluating all the available implementations of a service (service mirror) and selecting the one that is most suitable to the current context and user preferences (gives the highest utility).

The server planner is called by the adaptation manager and given the service mirror for a service that needs replanning. The service planner then acquires all the possible service mirrors for the service type of the service that is to be planned. A service mirror is fully resolved if it is atomic or all its dependencies are resolved. Before the planner can calculate utility for a service mirror it needs to resolve all its dependencies, so that it is fully resolved, also including these dependencies in the calculation. A service mirror that is not fully resolved cannot be used and thus will not be included in the planning phase.

The planner will calculate QoS predictors for all of the fully resolved mirrors by using the utility function that is provided in the *behaviour specification* of the service mirror. These QoS predictors give the expected utility that each of the implementations would give. The mirror which gives the highest utility is normally selected. If this is not the same service mirror as was running already, a reconfiguration of the application, changing the components as needed in accordance to the selected service mirror, is done.

The service planner has a binding to the context repository which will provide it with knowledge about the current context variables. These context variables are needed to calculate the utility of a service mirror given its context model.

2.1.9 Adaptation manager

The adaptation manager is in charge of controlling when a replanning of the running application is going to happen. The adaptation manager is often implemented by the application developer to make it work in the best way possible according to the needs of the application. There is, however, a very basic implementation of the adaptation manager provided with QuA.

The adaptation manager gathers information from the context monitoring components through the context repository and discovers if there has been significant change in context. If there has been significant change the adaptation manager calls the service planner which plans based on the behaviour specification of the running service. The service planner calculates utility based on the new context and reconfigures the application if needed.

For instance, the PMS application checks to see if there has been a substantial change in the bandwidth since the last planning of the application. This way, replanning is not triggered each time there are minor changes in available bandwidth.

2.1.10 Summary

The QuA middleware is vital to the functionality of the PMS application, handling adaptation of the video stream and the application. As the goal of this thesis is implementing session migration for the PMS application and it is hypothesized that each device can be represented as service mirrors in the QuA platform, an understanding of the inner workings of QuA and especially the planning and adaptation components and service mirrors are important. An in depth introduction to the current PMS application will follow in the next section. A more complete explanation of QuA and it's components can be found in [7, 3].

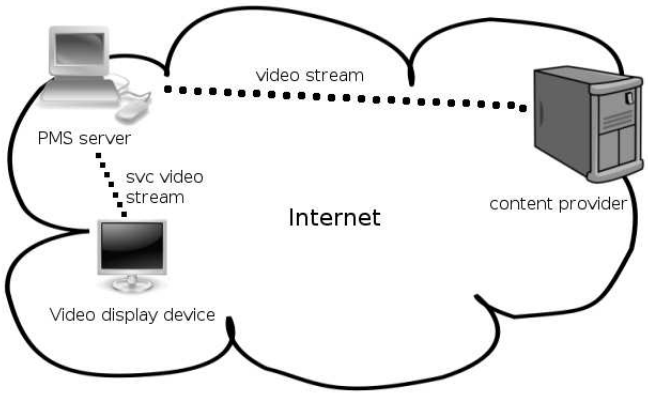

Figure 2.3: The PMS system

2.2 PMS

As the focus of this thesis is implementing session migration for the Personal Media Server application, a detailed knowledge of its architecture and implementation is important. Especially important is how PMS runs on, and interacts with, the QuA middleware, as it is hypothesized that devices can be represented as QuA service mirrors, and service migration being handled by QuA.

2.2.1 Overview

PMS (Personal Media Service) is an application developed to show an example of how the QuA middleware technology can be used for applications with adaptation needs [3, section 6]. PMS receives live video from an external feed and recodes it to a scalable video coding format. A client subscribes to a certain video feed and quality from the media server. Such a client could be any device which has network and video displaying capabilities, such as a laptop, cellphone or a television set(given that it has a network connection and a Java runtime environment to run the client application). Figure 2.3 illustrates the PMS system. An external content provider streams video data to the PMS, which transcodes the incoming video data to the scalable video coding format. This new data is then streamed to a device which displays the video.

The advantage of scalable video coding is that the video is coded in several layers of quality. This means that the client does not have to subscribe to the full video stream to get the video. However, to get the full quality, all layers are needed. A server that publishes the video stream encodes the video in the scalable video format and sends only the parts that each subscriber/receiver has requested. With the PMS application running on a personal media server in your home, you could receive a television broadcast over the Internet and have the broadcast forwarded to e.g. your television set or your PDA. Of course the PDA has a much smaller screen, and perhaps a lower bandwidth, so you would want a lower quality on the video stream to your PDA than to your 40" LCD television set. The PMS application handles this by transcoding the incoming video stream to scalable layered video, and each receiver could subscribe to whichever layers it would find relevant (dependent on user preferences or settings set by the application developer).

PMS uses the QuA platform and has set up several mirrors for handling a drop in bandwidth or an increase in delay. PMS has three ways of altering a video stream; *chrominance*, *luminance* and *frame-rate*. The preferences for these could either be hard-coded by the application developer, or set by the user. As of now, since PMS is still in the development phase and not available to any real users, the values are hard-coded into the application. The demo application also let you simulate bandwidth by means of setting it in a GUI.

Given the fact that PMS uses the QuA platform for adapting a video stream to the current context there is no need for the client to subscribe to a lower quality video if he believes that the bandwidth of the client device he is using is too low to handle the full quality video. As PMS monitors the bandwidth it will always stream the best quality video allowed for that bandwidth.

It could, however, be an advantage for the client to subscribe to a lower video quality if the displaying capabilities (screen resolution etc.) of the used device does not allow for showing the video at full quality. Since this kind of context is not implemented in PMS, video with a higher quality than allowed by the display might be streamed if the bandwidth allow for it.

If the bandwidth drops beneath the threshold for streaming even the lowest quality video, PMS will start to buffer the incoming video instead of streaming it to the client. This way the user will not miss any of the video broadcast, but will receive it at a later time if the bandwidth increases to where video can be streamed. When the bandwidth picks back up PMS will start streaming the buffered video to the client.

When streaming the buffered video to the client, time-shift can be used.

When time-shifting, the video is streamed at a different rate than its original frame rate. This way, by streaming the video for instance 1.2 times as fast, the buffer will eventually become empty and the video received by the user is again as close to live as possible from the broadcasting source. During the time-shift period the video will display at a higher frame rate on the client side. PMS makes sure that the bandwidth is high enough to compensate the extra video streamed to the client.

Time-shifting can also be used to slow down a video stream instead of stopping it completely when the bandwidth is too low. By streaming the video at for instance 0.8 the rate of the video frame rate it may be possible to still stream to the client with the available bandwidth. Video will show at a slower frame rate on the user side and lose some of the flow, but will not stop completely. The extra frames will be buffered up by PMS. If the bandwidth is then increased again, PMS can use time-shift to speed up the streaming to catch up with the live stream. This will give the user a more seamless experience than simply stopping the video when bandwidth is low.

There are several personal media server applications available on the market currently, such as Microsoft Windows Media Center, Ahead Nero Media Home and TVersity[1], which is a free alternative for personal use. Nero Media Home and TVersity does have transcoding capability (e.g. div-x to mpeg-2), however neither of them support scalable video coding or session hand-off.

2.2.2 Video adaptation and scalable video coding (SVC)

To adapt the video stream to the context of the client, PMS uses a technique called scalable video coding [2, 11]. Scalable video coding (SVC) is a technique to encode a video stream in separate layers of quality. To get the base quality a user only has to subscribe to one layer of the video stream. This will give the lowest video quality, but the user will be able to view the video. To get a better quality, the user can subscribe to the above layers which add to the quality of the output video. Each layer builds on the layers underneath. This means that to get the quality of a certain layer one also needs all the layers underneath, much in the same way that one needs the basement and the first floor to build the second floor of a building.

Even though PMS is designed to be a personal media system and is only meant to stream to one client at a time, SVC would also provide some advantages when streaming to multiple clients. As of today many video content providers on the internet has video encoded in several qualities and streaming them according to the bandwidth that is available to each user. By

using SVC only one copy of each video file would be needed, only transferring the necessary layers to each user.

Quality is divided into *chrominance*, *luminance* and *time* (frame-rate). A user can subscribe to these separately, and there are layers for each of them. The base layer of all three are required to render the video, but one could for instance have only the first layer for chrominance but all the layers for luminance. The time-layer indicates the desired frame-rate, subscribing to a higher layer giving a higher frame-rate.

Figure 2.4: Scalable video coding. Different quality layers.

Figure 2.4 illustrates the difference in quality with different layers of quality. Here, *ql* indicates the luminance quality layer, and *qc* indicates the chrominance quality layer. The top left picture has full quality, while the bottom right one has only the base quality (lowest quality). Difference in luminance quality is more visually apparent than change in chrominance quality giving the image a much rougher look. A drop in chrominance has more subtle effects, such as in colour detail difference and a reduction of contrast. The full quality gives a sharp, detailed image while the base quality gives an inferior

image, but one that still allows the viewer to discern the objects pictured.

SVC also provides the opportunity to divide a video into sections, and a user can subscribe to different qualities for each of these sections. An example of this could be if a tv-channel was broadcasting a sports overview event where an anchor person is occupying most of the screen, but they are showing different clips from sports events in the right corner. The user would probably be more interested in watching the clips in the corner rather than the person and would want a better quality for that. If a user was only interested in the clips he could subscribe only to this section of the video stream. If the user was watching the video on a hand-held device with a small screen this would be a valuable option to enhance user experience. This functionality is, however, not implemented in the PMS application.

As with many other video codecs SVC exploits the fact that consecutive frames in a video very often are similar. To reduce the amount of data needed to recreate a video stream each frame is encoded based on the difference from the last frame. This is known as *differential video coding*. Frames that rely on previous frames are known as inter frames. The first frame that is transferred needs to contain all the data to recreate the image. Such a frame, known as an intra frame, does not need any other information to be rendered correctly.

Very often, as is the fact with PMS, video streams are transferred using the UDP protocol, which is known to be unreliable. Packet loss is common when streaming with UDP, and missing a frame when encoded in this way would lead to a corrupted video when rendered without all the information. To compensate for this, or other issues that might occur, frames at regular intervals are encoded and transferred as intra frames. These intra frames ensure that a video stream is corrected if it has been corrupted. In the video codec used by PMS every eight frame is an intra frame.

A more detailed overview of scalable video coding can be found in [2].

2.2.3 The PMS Implementation

The current implementation of the PMS application is not really a distributed system as both the server and the client are running in the same process. However, all the communication between the two parts are done using UDP, so splitting the client and the server into two separate applications should be trivial. The code for PMS is written in Java.

Both the server and the client parts are running on the QuA middleware platform. Only the server part, however, has any need for adaptation (adapting video quality). The only function of the client part is receiving SVC coded

video data, decoding it and displaying it. Thus, the client part could easily be implemented not to run on QuA. Due to this, the focus of this section will be on the server part of PMS.

The initial service of PMS, *WatchTV*, did not support time-shift. This service was later evolved into the service *WatchTV-TS* which also supports time-shift and buffering. The *WatchTV-TS* type conforms to the *WatchTV* type, which means if QuA is running a service of type *WatchTV* mirrors with type *WatchTV-TS* would also be considered. However, the extra QoS dimensions of the *WatchTV-TS* type would be ignored [3].

The PMS main application code bootstraps QuA and advertises the WatchTV service mirror which is of type *WatchTV-TS*. All the components that are used by PMS are also advertised to the instance of the QuA platform, and the context repository and the GUI is setup.

Each of the three different configurations for PMS (compositions), live, storage/buffering and time-shift, are set up. In this setup process all the components that are used in each of the compositions are set up. The binding specifications between all of these are also set; these need to conform to each of the components interface.

The functional behavior of of the WatchTV service is to stream live video content to a client device. Its defined error dimensions are: temporal-, luminance- and chromance quality [3]. These are the quality layers of the realization of SVC as described in the previous section about SVC.

A description of the implementation of each of the components in PMS is given in section 2.2.7.

2.2.4 The live configuration

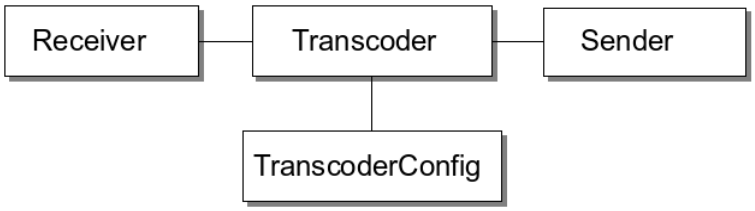

Figure 2.5: Live configuration of PMS.

The standard configuration of PMS is the live configuration. Figure 2.5 shows the architecture of the live configuration. The server receives an incoming video stream from a source through the *Receiver* component. A *Transcoder* component is running a thread which gets the video data from the receiver and decodes the incoming video data. The transcoder thread waits for a notification from the receiver, containing video data. The incoming video data can be encoded with any video codec as long as the transcoder has access to a component which is able to decode it into raw video data. The transcoder then encodes the raw data to a scalable video coding format. The quality of the video sent (what layers) is determined by the *TranscoderConfig* set for the transcoder. These layers are then sent to the *Sender* component which sends the data to the client.

2.2.5 The storage configuration

Figure 2.6: Storage configuration of PMS.

Figure 2.6 shows the architecture of the storage configuration. When QuA discovers that the available bandwidth between the server and the client drops below such a level that video data cannot be streamed it will reconfigure the application to use the *storage composition*. The *storage composition* is the simplest of the PMS configurations, containing only three components. The incoming video data is received by the *Receiver* and is then read by the *Storage* component. The *Storage* component then forwards it to the *Buffer* component which stores the incoming data to a buffer.

No transcoding is done in the storage composition, but the incoming video data is stored as it is. Transcoding of this data will start when the bandwidth picks up and the application is reconfigured to the time-shift configuration.

2.2.6 The time-shift configuration

The time-shift configuration is the most complex configuration of PMS. This configuration is selected if the application has been in the storage composi-

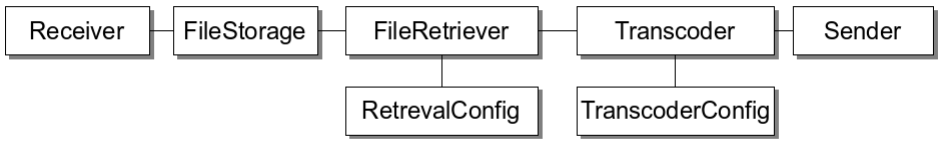

Figure 2.7: Time-shift configuration of PMS.

tion and bandwidth increases, allowing for streaming the video. Depending on user preferences, this configuration could also be selected if the bandwidth is to low to stream at the full frame rate and a time-shift to slow the video stream is desired.

The time-shift configuration is also used after storage to stream at the correct frame rate. When the application has been in the storage state and bandwidth is again at an acceptable level for streaming at a normal rate, the application still needs to stream from the buffered video or else frames would be lost. Some times one would like to time-shift by speeding up and sending more data to the client, but depending on user preferences and on whether the bandwidth allows for this increase in data sent, it might be desirable to stream the video at the regular speed.

During time-shift, incoming data always need to be stored to the buffer. The oldest data in the buffer is read and transferred to the client. Only when the buffer is empty will the application go back to the live configuration. This will happen if the buffered data is time-shifted and streamed at a higher rate to the user. Time-shift to storage reconfiguration can happen at any time if the bandwidth drops too low.

Figure 2.7 shows the architecture of this composition. It is a combination of the live- and the storage- compositions. Incoming video data is received and sent to the buffer. The oldest data is then retrieved by a *Retrieval* component at the rate specified by the *TranscoderConfig* component. This data is passed on to the transcoder which transcodes the data to the SVC format and passes it on to the sender which streams it to the client.

2.2.7 PMS component implementations

The following information which covers the current implementation of the PMS application is solely based on the source-code, as there is no documentation which covers this.

The current implementation of PMS does not receive the incoming video data from an external source but it is read directly from a local file. This makes testing and setup easier. QuA makes it easy to change this by simply adding to or changing the components used.

Each of the components used in the three configurations (compositions) of PMS are implemented through QuA components. This makes them easier to replace by simply developing new components that conform to the service type of the component to be replaced. For instance The *Receiver* and *Sender* components are implemented as simple UDP protocol sender and receiver. If one wanted to use a different protocol one could simply implement it conforming to the service type and interface and have QuA use it, instead. No other changes would be needed to the application. One could also have QuA measure, for instance, packet loss when using UDP and have it dynamically adapt the application to use a different protocol instead if it was too high. A brief walkthrough of the current implementations of the most important components in PMS follows.

The PMS implementation also use *Subscriber* and *Publisher* components to pass video data and messages, and handling the video quality. However, these will not be discussed here as they have no implication for the implementation done for this thesis.

The *sender* and the *receiver* are as mentioned implemented for using the UDP protocol (*NetUDPReceiver* and *NetUDPSender*). The Buffer component is implemented as *PMSFileBuffer*, which simply writes the information to a local file. The *Retriever* is then implemented to retrieve this information from the file (*PMSFileRetrieval*).

The *transcoder*, implemented as the *PMSDummyTranscoder* component, does not transcode the incoming video, as the incoming video data received in the test application is already in the SVC format. It simply passes forward the correct data received, depending on the *TranscoderConfig* which has information about what video quality should be sent, to the Sender.

The two configuration components (*PMSTranscoderConfig* and *PMSRetrievalConfig*) are a bit different from the other components used in QuA. This is because these are the components (service mirrors) used by QuA when calculating what composition to use. There are several different possible configurations of the *Transcoder* and the *Retrieval* components, each of these configurations have their own service mirror. When the service planner performs planning, it finds out which of these give the highest utility, given constrains and context, and selects them. These two components have no other functionality than storing configuration information.

The *PMSTranscoderConfig* component stores information about the quality layers for the video; *time*, *luminance* and *chrominance*. One *TranscoderConfig* service mirror is selected by the planner and set in the *transcoder*, which tells the *transcoder* which quality of video to send to the client. Only configurations with the same layer for chrominance and luminance are implemented. There is one service mirror for each of these configurations for each level of the time quality.

The *PMSRetrievalConfig* component stores information about the time-scaling used when the Retriever retrieves video data from the buffer. It also stores information about the temporal displacement. Three service mirrors for retrieval configurations are implemented in the current PMS application; 0.8, 1.0 and 1.2. A value of 0.8 time-shifts using a slower rate, 1.2 time-shifts speeding up the video. 1.0 retrieves at the rate of original video.

2.2.8 Adaptation and reconfiguration

An implementation of the adaptation manager has been made especially for PMS, and extends the core adaptation manager interface. The adaptation manager runs in its own thread monitoring the context that the system is running in, at regular intervals (every 500ms). It stores the context values for each of the QoS dimension each time a reconfiguration occurs. This way it can compare the current context to that to determine if there had been a large change in for example bandwidth. If there has been, the adaptation manager triggers the QuA service planner for replanning of the application.

Whenever a reconfiguration is to occur, the adaptation manager is responsible for making sure that the application is in a safe state before reconfiguration occurs. To do this it keeps a list of all the currently running components before a reconfiguration and compares this to the list of components given by the planner in the new configuration. The components that are no longer in the new configuration are then stopped in the correct order, making sure that no data is lost. When all components are stopped, the adaptation manager starts the components that have been added. Lastly the unchanged components are updated.

Calculating utility to select a service mirror is done through two functions; *PMSUtilityFunction*, and *PMSErrorEstimator*. The *PMSErrorEstimator* is a helper function to the utility function. The utility function of PMS can be represented the following way:

$$U(t, y, c, d, r) = W_t K_t(t) + W_y K_y(y) + W_c K_c(c) + W_{d,r} K_{d,r}(d, r)$$

where U is utility, W is weight and K is predicted QoS for each of the dimensions; t - frame rate; y - luminance; c - chrominance; d - temporal displacement; r - streaming rate.

The Error estimator calculates utility for four different error dimensions; frame rate, time scaling, video quality and time displacement. Frame rate, time scaling and video quality are calculated depending on the values given in the PMSTranscoderConfig and the PMSRetrievalConfig mirrors. A minimum needed bit-rate is calculated based on these values. If the minimum bandwidth exceeds the currently available bandwidth an exception is raised, which means that this configuration of service mirrors for the service will not be chosen by the planner.

If all possible configurations of transcoder configuration and retrieval configuration raises exception, live- and time-shift configurations of QuA are impossible, and storage/buffering will be chosen.

The values given by the *PMSErrorEstimator* function are then used by *PMSUtilityFunction* to calculate the total utility given by this configuration of service mirrors. This is done by applying weights set to each of the error dimensions and calculating an overall utility value. These weights represent user preferences. For instance having a weight of 3 for quality and a weight of 1 for the others will mean that the user thinks video quality is as important as the other three dimensions combined. The overall utility value is returned to the *service planner* which selects the configuration which gives the highest utility for reconfiguration.

2.2.9 Summary

A good understanding of the PMS architecture and its implementation as well as the underlying QuA middleware is important since the focus of this thesis is implementing session migration for the PMS application. It will be necessary to make changes to the architecture and some of its components in order to do this so understanding how they work is vital, especially the process of sending data to a device and the different configurations of the system, as session migration should also work in time-shift mode and not only in live mode.

2.3 Session migration

In this section the concept of session migration will be explained in detail, and an overview of some previous work done in this are will be presented.

2.3.1 Overview

In this thesis, the word *session* is used to describe a separable block of interaction between two nodes in a network. Session can be used not only to describe streaming of media, but also things like interaction between a web browser and a web server or between players in an online game. This focus explicitly focuses on video streaming sessions, but will also touch briefly on other types of media streaming sessions.

The term *session migration*, also known as session transfer, handover or hand-off, is used to describe the concept of changing one of the participants of a session with another. In this master thesis the kind of session migration done is changing the recipient of a video stream so that the video is continued from the same point on the new recipient. Figure 2.8 illustrates session migration occurring in a client-server video streaming system. The vertical dotted line shows the point where a session migration has occurred.

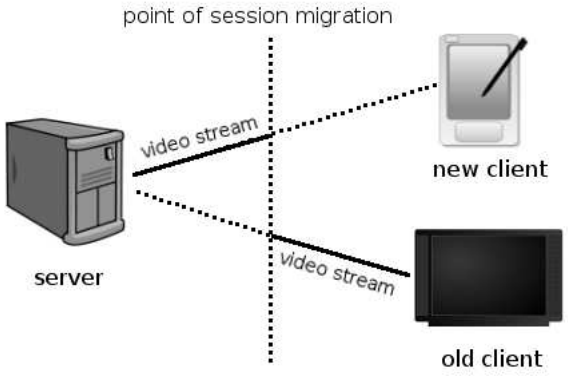

Figure 2.8: Session migration.

2.3.2 Motivation

Session migration is a very handy tool in some situations like the scenarios described in the introduction chapter. [13] gives three reasons for session migration; cheapest cost, better user experience and physical user mobility. The latter two are the most important motivations for this master thesis.

An example of cost effective session migration would be a user watching a video stream on a mobile device that has both UMTS and WLAN functionality. If the video is currently streamed over the UMTS network, and the user enters a wireless hotspot, the user might want to transfer the streaming

session to the wireless net since the cost of UMTS is much higher than that of wireless LAN, which are often free to use. When the user leaves the hotspot, the stream would be transferred back to the UMTS network. Although the user still watches the video on the same device, it is still considered session migration since the stream needs to me moved from one network topology to another which will assign the device different ip-addresses. This scenario is discussed in depth in [10] .

An example of session migration to better user experience would be transferring the session to a larger screen with better quality, such as an LCD television set. Physical user mobility would be to transfer the session from a TV to a hand held device, or another room to make the stream 'follow' the user.

[13] also divides session migration into 'complete' and 'partial' session migration, where complete session migration transfers the whole stream whereas partial may transfer for instance only the audio. In this case partial session migration can be seen as *splitting* the session containing both video and audio into two separate sessions; one for video and one for audio. Another kind of session migration would be session duplication, where a session is duplicated to provide the exact same data to the another device. An example of this would be duplicating a video streaming session so that the video continues from the exact same place on another device while still continuing on the originating device. More use-case scenarios for session mobility can be found in [6].

2.3.3 Session migration architecture

The architecture for session migration can be divided into three approaches; *device centric approach*, *network centric approach*, and a *hybrid approach* [13].

In the *device centric approach* all the session migration mechanisms reside in the devices. This means that there will be no changes necessary to the network infrastructure. However, this approach might also lead to more complex and expensive devices.

The *network centric approach* puts almost all of the session migration mechanisms in the network. The only role the device plays in session transfer is to advertise itself and request a session transfer explicitly. Session transfer may also be initiated implicitly by the network choosing the optimal device based on preferences.

The *hybrid approach* combines the two other approaches. Here it would be

natural to have the network responsible for device discovery and exchanging device capability information and leave the session initiation and transfer to the devices.

PMS is an example of *device centric* session migration architecture. The PMS server handles device discovery and also all the mechanics of migrating the session from one device to another. This is done based on context information collected from each of the devices. If the user explicitly select a device as the active one, this device will then have to inform the server which migrates the session to this device.

2.3.4 Session migration in server-client architecture

In a server-client architecture, session migration can be divided into two categories; client side and server side session migration. Server side session migration is by far the most common, and much research has been in this area. Server session migration is essential in mobility. For instance person talking on a cellphone while in the car may need to change from one GSM access point to another, and session migration is performed. GSM is an example of network centric session migration where session migration is handled by the network and is transparent to the device. This master thesis focuses explicitly on client side session migration, as it is the kind of session migration that is to be performed by PMS.

The next sections will present in detail some of the research previously done in these two categories of session migration. Even though this thesis is about client side migration, server side migration will also be covered as some of the concepts are similar, and as mentioned, much more research has been done for this kind of session migration.

2.3.5 Server side migration

Server side session migration is different from *client side session migration* in that the server in a session is changed. This is a very common concept in mobile networks and mobile telephony. An example of this could be a user at a university campus using a laptop which is connected to a wireless local area network (WLAN). If this user were to move his laptop to another place on campus out of the bounds of its current access point (WLAN router), but where another access point is available, session migration from the first access point to the new one should be performed. If the availability of these two wireless hotspots overlap this should be done without the user noticing anything, even for time sensitive applications e.g. VoIP.

2.3.6 Example: Mobile media transcoding sessions

The PMS server is a server transcoding a video (either an external stream or a local file) to a scalable video coding format. [16] proposes a different approach to streaming multi-media to devices with different capabilities and context, such as hand held devices. A multi-media stream is transferred from a central content provider. At servers closer to the end user the video stream may be transcoded to a stream with lower quality to better comply with the needs of its users. However, when a mobile user moves out of the bounds of the server that is transcoding its video stream, the transcoding session has to be handed-off to a different server that is in the vicinity of the user device. For this to work and to make the session migration smooth (smooth handover) so that the session migration is not noticed by the user the transcoder state needs to be transferred from one transcoding server to the other.

In this example we have session migration from server to server, and not from client to client as we are trying to do in PMS. The servers in this other approach are not personal, but offers a different approach to streaming media to devices with different capabilities and context. Figure 2.9 illustrates this approach. The servers labeled 'server 1' and 'server 2' represent the transcoding servers.

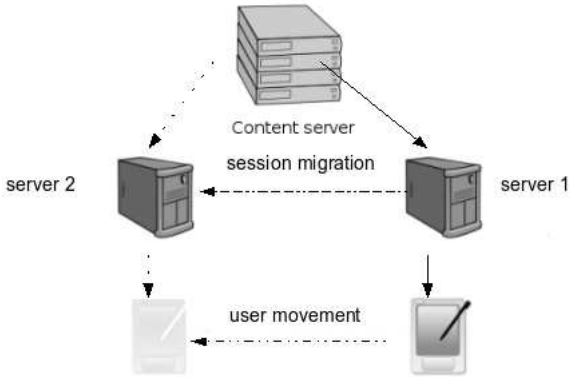

Figure 2.9: Server session migration.

This solution is designed to handle device mobility whilst the idea behind PMS is handling user mobility and migrating between devices. Even though this approach is different from the PMS approach, they have many aspects in common. Both need to handle smooth hand over and consider transcoder

state, due to differentially encoded video. The difference lies in that this approach migrates a session from one transcoding server to another and the device stays the same. If the transcoding servers would just display the video streams instead of transcoding them and sending to the user we would have a more similar approach to that of PMS. Then the transcoding servers would in essence be clients and we would have client side session migration. An in depth presentation of this approach can be found in [16].

2.3.7 Client side migration

In a client-server architecture, where the client receives and exchanges information with a server, the term *client side session migration* is used to describe session migration where the client (device) is changed. This is the case for the session migration functionality addressed in this thesis for the PMS system. Video is streamed from the server to the client(device) and when certain criteria are met the video stream is changed to another client(device).

2.3.8 SIP and session migration

Setting up and tearing down media streaming session are often done by using the *Session Initiation Protocol* (SIP)[5]. SIP was initially mainly used for voice over IP but can also be used for controlling other types of media streams as SIP is purely a signaling protocol and is independent from the media encoding and the protocol used to transfer the data. This protocol can be extended to support session migration as well. A presentation of how this is done in [8] is given in the following section.

When using SIP for session migration, the migration part is done by the devices, but is controlled by a server (Service coordinator). The decisions that a session migration is to occur can both be made by the service coordinator and the client device, depending on the implementation. The service coordinator is not necessary if the clients have means to discover devices and know how to communicate with them.

When a session migration is to occur, the current client receives information about the identity of the new device. The old client will then notify the new client about the session migration using the command SIP_REFER and including information about the state of the session. The new device will then try to establish a new session with the server informing it about its constraints and capabilities as well as the state of the session to be handed over. When this done, the new device will notify the old by sending a SIP_NOTIFY message. The old client will then close the connection with

the content distributor, and the handover is completed.

To correctly hand over a session from one device to another the content provider might need to adapt the DI (digital item) that it is currently streaming to the capabilities of the new device. To do this it needs the information received from the new client describing the session and the device constraints and preferences. This is proposed done using a combination of *Session Description Protocol* (SDP)[9] and *MPEG-21*[4].

2.3.9 Summary

Session migration is at the very center of this master thesis. Although not much work has been on the type of session migration of PMS, these other cases provide a basis and ideas for how session migration can be implemented. More extensive research about session hand migration can be found in [13, 10, 6, 17].

2.4 Smart Homes

The concept of session migration for PMS is closely related to the concept of Smart Homes. This section will give a brief introduction to this concept.

Smart homes are often referred to as automated homes [15, 12]. The motivation behind smart homes is to make the users life easier and more comfortable. This is done through adapting the home to the current context and the users needs. An example could be turning on and off the light as the user enters or exits a room, or have the refrigerator monitor its contents and automatically order groceries over the internet when something is needed, or inform the user of products that have passed the expiration date. This concept could be extended to all the parts of a home.

A prerequisite of smart homes is smart devices. By smart devices is meant appliances which have processing and communicating capabilities. Another prerequisite are sensors which monitors the current context and the user activities. In this way the house can react to the current environment. This is known as context-awareness [15].

The most obvious user activity is movement. By monitoring the users whereabouts the home can adapt to bring the user the services that he requires. This is the case of session migration in PMS which should move the video streaming session according to the users position.

In order for the home to react in the correct way, and all the devices to be aware of the current context, all the devices and sensors need to have communication capabilities. This could be done most easily by using wireless networking, e.g. WLAN or BlueTooth. An in depth presentation of this can be found in [15].

Chapter 3

Requirement Specification

This chapter will first present some scenarios where session migration would be useful. Then the requirements for the PMS application which supports session migration, from now on only referred to as *PMS*, will be presented. These requirements are based on the goals presented in the introduction and the result of the research done in the background chapter. These requirements provide the basis for the design which is presented in chapter 4.

All of the requirements given below are represented by statements which contains either the word must or *should*. Requirements which are specified by the word *must* are absolutes that must be fulfilled in the final implementation. Requirements specified by the word *should* are requirements that one should strive to fulfill when creating the design and implementation, but are not vital to achieving a working solution for the problem presented in this thesis. These are aims to increase the performance of the final implementation, thus giving a better user experience.

3.1 Desired situations

There are several situations where session migration in a live video streaming session might be a desired functionality. This section will give a brief introduction to some of these scenarios.

1 A user is receiving a live television news-broadcast from a tv-channel through the internet and his set-top box/personal media server to his television set in his living room. A particularly interesting news story is running, but he has to leave for work or else be late. He will still want to watch the

news broadcast. It would be desirable for him to be able to transfer the streaming session to a hand held device or a LCD screen in his car and continue displaying the live broadcast from the same point on the new device. This way the user could continue watching the news in his car (note that you should keep your eyes at the road at all times when driving a car) or on his hand held device on the bus or the train.

2 A child is watching a movie in the backseat of a car while traveling. This movie is streamed from the personal media server at home, over the internet and to the car using UMTS. The car stops at a gas station which provides free-to-use wireless internet connection. In this situation it would be desirable that the stream is migrated from the UMTS network to the wireless network, because this is much cheaper.

In this scenario, where session migration is motivated by cost, the session is not migrated to a different device, but a session migration will still need to occur as the receiving device will have a different IP address on the new network.

3 A user has a very modern home where there is a device in each room which can display video and is connected to a local server. The home also has sensors in each room which tells the system where the user is at all times. Using this information a user might want a video session to follow him wherever he goes in his home. For instance if the user exits the living room and enters the kitchen he will want the video displayed in the kitchen rather than in the living room.

4 Two people are watching a movie in their living room. There is only five minutes left of the movie, but one of the people watching has to leave to catch the last train home. This person naturally wants to watch the ending of the movie, but cannot stay any longer. The other user, however, is staying, and wants to continue watching the movie on the large television screen with high quality. In this situation a solution could be to duplicate the current video stream to a hand held device so that the video continues on both devices. This way both users can watch the end of the movie.

5 A user is sitting in his living room watching a movie streamed from his personal media server on his 40" plasma screen. However, the media server detects that he has a cellphone in his pocket which has video rendering capabilities. In this situation we do not want a session migration to occur, because the user would obviously want to watch the video on this high quality

television set rather than his cell phone, even though the cellphone is closer. In this scenario other context than just distance between user and device is necessary to select the device the user prefers.

3.2 Functional requirements

This section describes the functional requirements of the session migration functionality.

3.2.1 Consistency

To provide a reliable service the PMS application will have to be consistent when planning for which device to select given the context. This means that PMS must select the same device each time if the given context is identical, unless learning is applied in e.g. QoS prediction.

REQUIREMENT PMS must be consistent when planning for the given context.

3.2.2 Uniqueness

In order to select the correct device to stream to, PMS needs to have a way of identifying each device and its context. Thus a unique ID is necessary to separate each device from the others.

REQUIREMENT Each client must be identifiable by a unique ID.

3.3 Non-functional requirements

This section describes the non-functional requirements of the session migration functionality.

3.3.1 Scalability

As the name PMS (Personal Media Service) implies, PMS is a system that is intended for personal use. With this follows that the application does not need to be able to handle hundreds of thousands of devices. However, it should be able to handle the number of device that would be reasonable

for a private person to own. Today people seldom have more than 5 or 6 devices which have video receiving and displaying capabilities. However, this number will most probably grow in the near future, as ubiquitous computing is becoming more commonplace and the price of hardware decreases. To accommodate for this it would be reasonable for the PMS application to be able to handle tens of devices without any decrease in user experience.

REQUIREMENT PMS should scale to tens of devices without great reduction in performance.

3.3.2 Smooth handover

When streaming differentially encoded video, inter coded frames are dependent upon intra coded frames. If session migration is performed at an inter coded frame instead of an intra coded, the video displayed at the new device will be corrupted until the next intra coded frame arrives. To provide seamless and smooth handover, session migration should be done at intra frames. A session migration/handover which is done in such a way that no data is lost or corrupted will be referred to as *smooth handover*.

The time taken for a session migration is essential to the user experience. If a user moves from one room to another, triggering replanning in PMS, and the time taken to reconfigure and migrate a session is very high, the user would loose much of the video that is streamed. It is therefore essential that time taken for a session migration to occur is not high. Especially if the user has explicitly requested to receive the video stream at a certain device. The time taken for a session migration should be sub-second or at most a few seconds.

REQUIREMENTThe PMS session migration functionality should handle smooth handover.

3.3.3 Availability

Clients may want to connect to the PMS server at any time. This should be possible even if replanning or reconfiguration is currently being done and PMS is busy.

REQUIREMENT Clients should be able to connect to PMS at all times when it is running.

3.3.4 Robustness

As the connection to clients might be lost at any given time, PMS needs some way of discovering this. How this is done is highly dependent on the protocol used for communication between server and client. If lost connections are not discovered PMS might select to stream to devices that are no longer available. Planning for devices that are unavailable will also use unnecessary computation resources.

REQUIREMENT PMS must be able to detect when clients are no longer available.

Chapter 4

Design

This chapter presents the design of session migration for PMS, including other changes to the current implementation required for implementing session migration. This design is the foundation of the implementation to be done to test the hypothesis that session migration can be realized using the QuA planning based middleware.

4.1 Overview

The main task is to evolve the existing Personal Media Server (PMS) so that it allows for having knowledge of several clients (devices) at the same time and being able to perform session handover between these given some properties. The term client is used to describe the client application which is to runs on the device, receiving and rendering the video.

The way the existing PMS test application is structured, both the client part and the media server parts are running in the same process. This means that the PMS is in no way distributed and not really a client/server system even though the two parts only communicate using UDP. The first task for achieving the functionality of the desired application is to separate the client part from the server part, allowing them to be run on separate computers/devices. This task allows for more freedom in how the client application is to be designed. This will be discussed in detail in the following section.

The server application needs some way of sensing that new devices with the ability to receive video streams are within range and are able to connect to the server. QuA has no way of doing this context sensing. A way to do

this will have to be implemented or simulated. For this master thesis, this will be done by having the client application on the device doing an explicit connect to the server by sending a message. A better solution would be having the server automatically detect devices without any user interaction at the device. Device discovery is a complex subject and will not be handled in this master thesis.

The idea behind the PMS application is using QuA to dynamically adapt the video stream to the current context. This context will be extended to also contain information about the currently available devices and having PMS automatically choose which of these that yield the highest utility for the user. Such context can be client bandwidth, delay, or other QoS aspects, or it could be user preferences, such as the user selecting which device the stream is to be sent to.

The users location in relation to the location of each of the devices, as well as the devices' available bandwidth, will be the context that is used to automatically choose which device to stream to. It would be natural to stream to the device that is closer to the user. If the user is sitting in the living room the video should be displayed on the television in the living room, and not the one in the bedroom. And if he moves out of the house the video streaming session should be transferred to his hand held device even though his television at home has a higher bandwidth.

The following sections will describe in detail the design, and the reasoning behind the choices made, of session migration for PMS using QuA and service mirrors.

As was mentioned in section 1.3 signaling and communication between the server and the client application will be done through simple commands using the UDP protocol.

4.2 QuA

As it stands, both the server and the client are running on the QuA platform. Depending on what functionality is intended for the client application and the client device capabilities, such as memory, processing capabilities and battery power, it might or might not be desirable to have the client application running on the QuA platform.

If the client's only functionality is receiving, decoding and displaying the video it would be practical to have the client application running without QuA, since QuA consumes much resources and this kind of client would have

no need for dynamic adaptation of the application.

However there could also be reasons why one could want to have QuA functionality on the client side. For instance it would be desirable to use QuA's functionality for handling context. There are a number of context variables that could affect how the application should function. The current version of the PMS server only has one choice for video encoding, but others could easily be implemented. In such a situation it might be feasible that if the client device was running low on battery power, or weak connection, it might want a different video encoding that takes lesser toll on the devices resources. It might also be desirable to have different protocols for communication and transferring the stream so that the application could automatically reconfigure using QuA to select the one that best fits the current needs.

Another example of context that one might want to affect the client application is the level of light in the room where the video is displayed. One could use QuA to sense the intensity of the light and dynamically adapt the display's brightness, since this might be easier on the eye and also consume less power.

There could also be a hybrid solution of running the client application with or without QuA. QuA has functionality to have external QuA core running in another application, but having all the adaptation done in the main QuA instance. Such a solution could have QuA also running on the client without the decisions taking and adaptation mechanisms, but sensing client context and handing this over to the main QuA instance. This has not been thoroughly tested and might not be stable, and there is also the question of how to make this solution work with an ad hoc system where clients connect and disconnect without any kind of warning. Due to the complexity of this solution, and the fact that this QuA functionality might not be stable, the choice has been made not to implement the system in this way, which would put more emphasis on QuA than on PMS and session migration.

The use of QuA, context sensing and adaptation on the client side can be made very complex, and this is not really the focus of this thesis, so only a simpler design for the client application, running without QuA, has been chosen. This solution also fits the scenario where the client is thought to be able to run on a device with low resources, like hand held devices.

PMS has three different compositions which allow for different kinds of streaming; live, time-shifter and storage. It should be possible to handle session migration regardless of which of these compositions is currently employed by PMS. This means that if PMS is in the storage state and a new device with better bandwidth is discovered, the session should be migrated to this device, and PMS should be reconfigured to the time-shift composition.

4.3 Context

Session migration for PMS will be handled automatically by QuA using the context of the user and the devices. The available bandwidth of each of the devices decides what is the best possible quality that can be streamed to that device at the time. This gives a good indication of which device is most suitable to receive the video. However, this might not help the user, as a device which is in a completely different place than him might be chosen and he can not see the video.

Using the context of the user's and each of the devices' physical location for calculating utility and selecting device would solve this problem. By selecting the device that is always closest to the user it would be more likely that the user is able to view it. But, as in one of the scenarios in section 3.1, this might not always be the best, as a device that is almost as close to the user might have better capabilities for showing better quality of video. A combination of the context of device bandwidth and of location could be a solution to this problem. QuA allows for weighing utility against each other when calculating. In this design the proposed weight ratio between distance from user and max video quality is 1:1. This way they both are equally important. A device with very low possible quality but close to the user would not be selected over one that is slightly further away but is capable of better quality. This weight ratio might not be the optimal, but finding the optimal would take a lot of testing, and is dependent on the user preferences. This ratio could also be left as a choice for the user.

4.4 Location awareness

Using knowledge about user and device positions it would be possible to migrate a video stream so that it is always sent to the device closest to the user. This can be modeled at several layers of abstraction.

The simplest solution would be using the user and devices' physical position and transferring the stream to the device that is closest to the user geographically. This method needs no setup to work but requires that the server, responsible for controlling the session and transferring it, has knowledge about the geographical coordinates of the user and each device. These coordinates could be obtained by equipping the user and devices with a GPS receiver unit. It should be noted that GPS receivers do not work well indoors. Calculating the line of sight distance between the user and the devices would then be trivial.

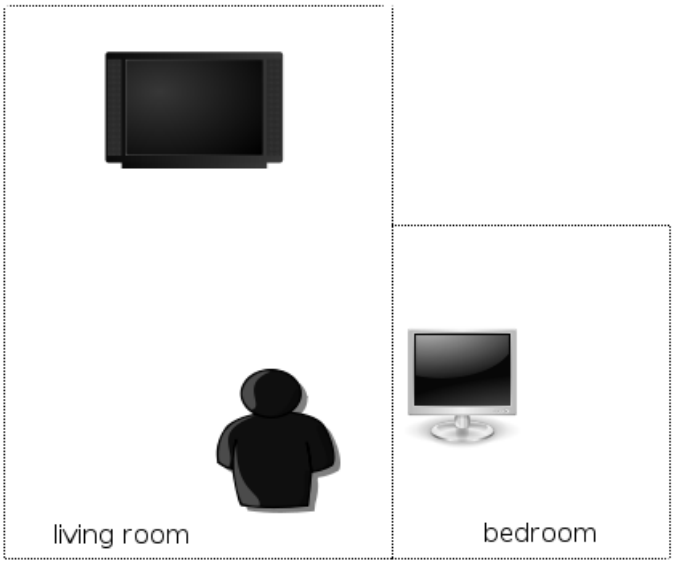

Figure 4.1: Location awareness: Problem situation

Advantages of this method is as mentioned that it is simple to implement and that no setup is necessary. However, this solution has some disadvantages. If the user is sitting in his living room and want to watch the live new broadcast on his television set a couple of meters away, the stream might not be sent to the television device if there is a device closer to the user. If for instance there is a device in the next room that is closer to the user, this simple scheme of calculating which device to stream to would select the device in the other room, rendering the user unable to see the video stream at all. This situation is illustrated in figure 4.1, where the user is in the living room, but closer to the device in the bedroom than the one in the living room.

Another scenario would be if the user has a mobile phone, able to receive the stream, in his pocket, this device would also be selected in stead of the preferred television screen. This problem could be solved by letting the server also take device capabilities into account, such as screen size, screen resolution and bandwidth. This way the television set which obviously has better capabilities for displaying and receiving the video stream would be selected rather than the closer mobile phone. As described in the previous section about context, only location and bandwidth will be used as context information for the implementation in this thesis.

The problem where the device in the next room is selected cannot be solved

as easily. A higher level of abstraction of the context model would be needed when considering user and device location. By modeling the rooms of a house and discovering which room the user and the devices are in, the server could simply select the device which is in the same room as the user, or the room closest if no devices are in the same room as the user. For instance device locations could be ”living room”, ”office”, ”kitchen” or even ”outside”. The location of the user would have to be represented in the same way. If a user has a device in each room of his home he could have the video stream migrate so that it is always displayed at a device in the room he is in. By representing location in this way, calculation of utility could be skipped, and instead filtering could be used on the location of the device being the same as the users location. This way utility would only be calculated for devices at the users location. If several devices are in the same room as the user, device capabilities could be used in a similar way as mentioned for the first method to select the most suitable device. This solution would give a better user experience since the scenario with devices that are closer to a user, but in a different room, will no longer be a problem.

Discovering which room a device is in could be done in different ways. One way would be defining the boundaries of each room using geographical coordinates, then finding which room the device is in. This is not a very good solution as it requires defining the rooms before it can be put into use. Each room would have to be defined by its geographical position and layout. This would be difficult to model due to the fact that not all rooms are rectangular in shape. GPS positioning systems are not always accurate and the position could easily be calculated to the wrong room, and GPS receivers do not usually work indoors. This approach also involves a lot of calculation on the server for locating each device and the user.

A much better approach would be having sensors in each room discovering what devices are in the room and if the user is in the room. This would give and easy and accurate indication of user and device locations and the server could simply filter out all devices that are not in the same room as the user. Setting up this system would require a sensor in each room which is able to communicate with the server and might be an expensive solution, depending on the price of the sensors..

For scenarios that are a bit larger in scale than an apartment or a house such as an office building with several floors a hybrid solution might be used. For instance one could get a good indication of a devices geographical position by what means it is connected to the network. For instance a wireless device probably would connect to a different wireless router for each floor of the office building. By this information devices on all the other floors could be filtered out.

The abstraction of device and user position could be made arbitrary complex. The simpler version using geographical location will be used in this thesis. This solution scales better and needs no set up of different locations, but can rely solely on the geographical coordinates. One would only have to calculate the straight line distance between two points, which is a simple geometrical calculation. This solution will also be used to show how the utility function is used (without filtering) and is easy to visualize.

4.5 Smooth handover

Smooth hand over is one of the main aspects to consider when dealing with session migration of streaming applications. A handover (migration) is considered smooth when no data is lost or corrupted, and handover is done in such a way that the user will not notice it.

In the migration of a video stream non-smooth hand over would decrease the users experience, as video data will be corrupted. For other streaming applications smooth hand over might be vital if they rely on correct data.

The video codec used in the PMS implementation uses the redundancy from one picture frame to the next in a video to reduce the amount of data in the resulting encoded video. The video codec uses some key frames (intra frames) to decode the following frames (inter frames) in the video stream. If the data of this key frame is not available to the video decoder, the resulting frames will be corrupted. The video decoder currently used by PMS will not crash or give any exceptions but will only display these incorrect frames. They will be displayed as the difference of the specific data encoded for that frame. This results in an image that often looks similar to a negative.

To achieve smooth hand over of a video streaming session, the video decoder at the new device needs to receive all the data needed to render the first frame of data correctly. This means that the first frame received from the sender needs to be a key frame or the new video decoder needs to receive, and be initialized with, the state of the old video decoder.

The following sections will present, and discuss, alternative solutions for handling smooth handover for the PMS application. Then the choice made for this implementation, and the reasoning behind it, will be presented.

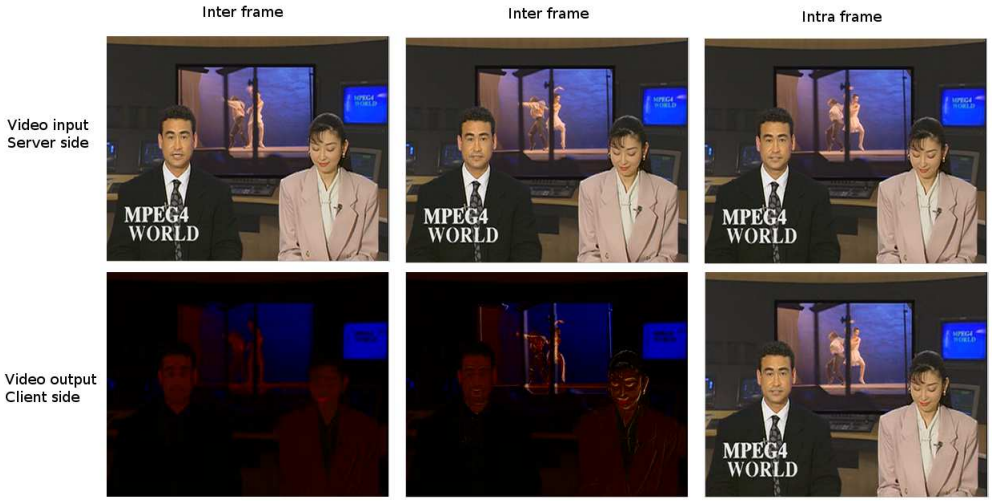

Figure 4.2: Non-smooth hand over

4.5.1 Delayed decoder initialization solution

A solution to the problem of smooth handover would be to let the new video decoder wait until it receives a key frame before it starts feeding the rendered frames to the video sink. To achieve something similar to smooth handover, using this approach, the video stream would also need to be transferred to the old client for some time to assure that the video stream is displayed at one of the client devices at all times. This might not be a preferable solution depending on the servers resources, mainly the out band bandwidth.

4.5.2 Decoder state transfer solution

The suggested solution where the video decoder state is transferred from the old client to the new client might be difficult to do. This is because a video stream frequently sends new frames which might change the state of the video decoder. This might lead to having to pause the process of decoding and sending the video for a session migration. Since all frames received at a video decoder might change its state, it is essential that the state of the decoder is received at the new video decoder before the next frame arrives. This transfer of state would probably take more time than the interval between two frames in the current PMS system, the state having to be passed from the old client to the PMS server and then to the new client. To pause the stream would introduce the need for a buffer in PMS. For this

solution to work the buffered video would have to be streamed at a higher rate than normal to empty the buffer, else the buffer size would only build up for each session transfer.

4.5.3 QuA smooth handover solution

A good solution for achieving smooth hand over would be having the hand over happen at the exact moment when the next frame to be sent is an intra frame. This way no video decoder state would have to be transferred from the old decoder, and there would not be any need to pause the video encoder or introducing a buffer.

However, this solution is not easily implemented when using QuA. The session migration functionality would have to reconfigure the PMS application at the exact correct moment as the video encoder is about to send an intra frame, which means that the underlying architecture would need some way of knowing when this happens. QuA currently has no way of controlling when a reconfiguration is to occur. The video decoder would need some means of flagging the moments when a session migration is possible, as perhaps a context variable or trigger the session migration in the underlying architecture. The time available for transferring a session correctly is not very large (for a video running at 25frames/s only 40ms separate each frame), which would put more limitations on the architecture.

A mentioned, the QuA platform has no implemented way of achieving this. When a application reconfiguration is triggered, utility calculations is done for each service mirror based on the current context. When the utility calculations are complete, the service mirror which achieved the best utility score is selected and application reconfiguration is performed. The utility calculation takes some time, and if checking context for the appropriate time for reconfiguration (intra frame) is done, the video encoder could very probably have already passed on to the next frame (as it runs in a separate thread) and session hand over would not be smooth. As QuA currently has no way of pausing the reconfiguration after the utility calculation and controlling when to do reconfiguration, it is not possible to implement this solution without altering QuA.

4.5.4 The selected solution

The two solutions presented above could provide some semblance of smooth handover, but are highly complicated to implement, and the little gained for implementing them would not be worth the increased complexity of the im-

plementation. Therefore, neither of these two solutions will be implemented for PMS for this thesis.

A "workaround" solution where QuA is not used for achieving smooth handover will be used. When planning has finished and has initiated a session migration by changing the device which is to receive the stream, the reference to this new device will not be used by the transcoder until the next intra frame is to be sent. The transcoder will use the old reference until an intra frame is reached, even if QuA has set a new reference. This way changing from streaming from one device to another will be instantaneous.

4.6 Preferred device

The user of a device will have the option to explicitly select that device for receiving the video. This will from here on be referred to as setting the device as the *preferred device*. When a device is set as the preferred device, no other devices will be considered when planning. When the preferred device is no longer available, or the preferred device status has been removed, planning will commence as usual, and the other devices are considered.

4.7 The PMS client

The PMS client application will be implemented as a simple client not running on the QuA platform. The client contains a *UDP receiver*, *UDP sender*, a *video decoder* and a *video sink*. As the client is split from the rest of PMS, as many of the components used in the original code as possible will be used. This way the client could be changed to use the QuA platform without to many changes.

The client application should be able to connect to the PMS server letting it know that a new potential client/device is available, thus subscribing to the service PMS offers. The server should provide the client with an identification key. The user of the client application will also have the option to set that device as the *preferred device*, having the PMS send the video stream to that device regardless of other context variables such as bandwidth. It will also be possible for the user to remove the device as preferred, letting PMS' utility function select which device is the optimal. If a client sets a device as *preferred device* PMS will stream to that device regardless of whether another device was already set as preferred.

The client application will also have functionality for letting the user discon-

nect from the PMS server (unsubscribe), removing the device from the pool of devices available to the PMS server.

4.7.1 Client Components

This section will provide a brief overview of the components used in the client application. As mentioned, these are the same components that were used in the original PMS application where both server and client was implemented to run in one process. Figure 4.3 shows the architecture of the PMS client application.

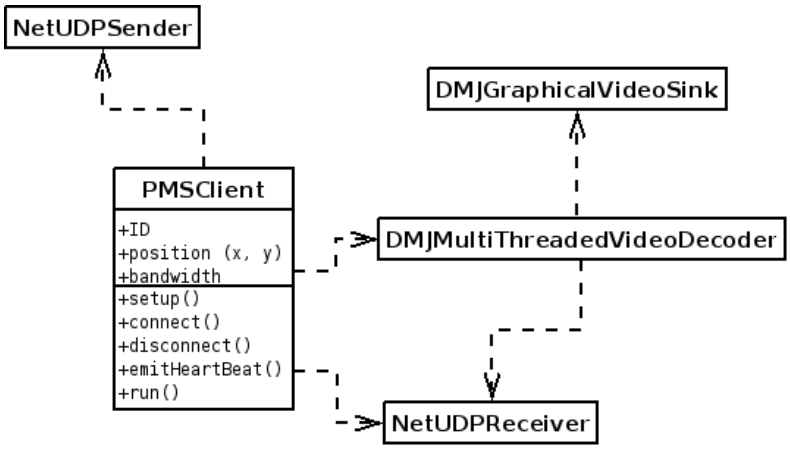

Figure 4.3: Client application - UML diagram

The *UDP sender* is responsible for sending information to the server. Information that is sent is users requests to the server, such as request to connect to a server, disconnecting and setting the device as *preferred*. It will also send information about client context, this will be piggybacked on the heartbeat and connect messages. Since UDP is not connection oriented the client needs to send a heartbeat message at regular intervals to enable the server to discover when the connection to a client has been lost without it having sent a disconnect message. An overview of the different commands the client should be able to send will be presented in the next section.

The *UDP receiver* is responsible for receiving data from the server. At the startup phase, when a client connects to the server, the client will receive data that is relevant for the session. After this initiation phase only video data will be received by the client. This video data is forwarded to the *video decoder*.

The *video decoder* is set up to subscribe to a certain video quality from the

server, and receives the encoded video data from the *UDP receiver*, decodes it, and forwards the raw video data to the *video sink*. This video quality is the optimal quality requested and since PMS will handle adaptation of the quality to the current bandwidth, there is no reason not to set this to the best quality.

The *video sink* displays the raw video data, which was received from the decoder, in a window on the client device.

4.7.2 Session control

This section will present the different commands that are available for controlling the session and the devices status through the client application. The commands to the server for each of these will be defined.

Connect

When connecting to the PMS server the client application will first send a connect message to the server containing the port number on which the user wants to receive the video stream as well as the client devices context (bandwidth and position). It will then wait for a response message from the server containing the devices ID as assigned by the server. The video decoder, video sink and the UDP receiver will then be initiated for receiving and displaying video.

Disconnect

When disconnecting from the server, a disconnect message containing the clients ID is sent to the server. The video decoder is then stopped, which will also stop the video sink and the receiver threads. This will also be done if the user shuts down the application, disconnecting from the server before exiting.

Preferred device

When the user has requested that the device should receive the stream (be set as preferred device) through the applications user interface, a message will be sent to the server telling it, rather than asking, for it to set the current device as the preferred device. This message contains the 'pref' command as well as the clients ID which was received from the server at connection.

Free stream

The free stream command works in much the same way as the preferred device command, but clear will clear the preferred device status. A message is sent to the server asking it to "free" the stream. The client ID is not included in this message. The ID is not necessary since any client has the ability to remove the preferred device status at any time regardless of whether it currently is the preferred device, or if there even is one.

Heartbeat

At regular intervals, the device will emit a heart beat. The heartbeat is a message that is sent to the server to inform it that the client is still alive. The message includes the clients ID and is also used to update the server with the current context of the client device, so bandwidth and position is also included. The heart beat message is used by the server to discover if the connection to a client has been lost, if for instance the client application has crashed.

4.8 The PMS Server

The handling of video in the PMS server application is kept as in the original PMS code. The only change is that the *video transcoder* which transcodes the incoming video data to a scalable video coding format no longer have one sender object, but will have information about one PMSClient object, which represents a device, which in turn has information about what sender object to use. Which of the available clients that is currently active is transparent to the *video transcoder* but is handled by the QuA platform and the adaptation manager. The application will be reconfigured by QuA and the proper device for the current context will be set in the *video transcoder*. The video transcoder will not stop for this reconfiguration. As mentioned in the section about smooth handover, the actual change of devices will be delayed by the transcoder until an intra frame is about to be sent. When the reference to the new PMSClient object is set, it will seamlessly start to transfer the encoded video data to this client. The PMSClient objects will be represented by service mirrors which are used by QuA for planning and reconfiguration.

PMS should be able to handle session migration regardless of which configuration that it is currently using; live, time-shift or storage. By using planning in QuA this should be done automatically and QuA would recon-

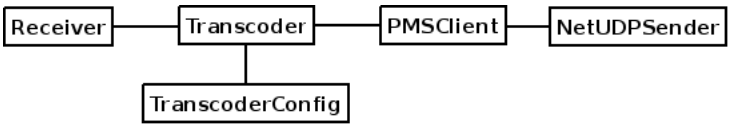

Figure 4.4: PMS live configuration with session migration.

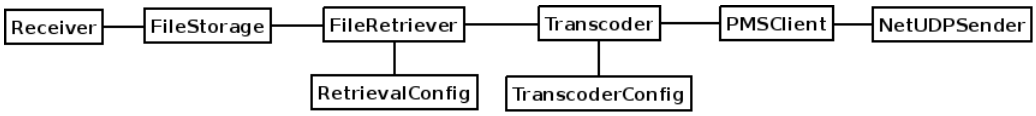

Figure 4.5: PMS time-shift configuration with session migration.

figure the application to another state if necessary. For instance, if PMS is in the storage configuration because no device can receive the stream, and a device that can receive is discovered, QuA should automatically perform session migration and change to the time-shift configuration.

4.8.1 Evolution of PMS

The focus of this thesis is evolving the PMS application to also include session migration. This involves some changes to the PMS architecture and configurations are needed.

Session migration is only relevant for the live and the time-shift PMS configurations. Figures 4.4 and 4.5 shows the new versions of these two configuration architectures. Explanation of the old configuration architectures as can be found in sections 2.2.4 and 2.2.6.

The difference between these configurations and the old ones is the *PMSClient* component between the *Transcoder* and the *NetUDPSender* components. The PMSClient component is realized by a service mirror for each device and represents the device and information about it. When a session migration occurs, the reference to the PMSClient in Transcoder is changed.

The evolution of the PMS application also involves a new dimension in the QoS error predictor and changes to the utility function, using the distance between the user and the device. Context sensing for device discovery and other device context will also be added. This will be covered in detail in later sections.

4.8.2 Handling context

PMS and QuA uses a context repository for storing context information. A context element, in form of a string, will be added to this repository for storing information about the status of *preferred device*. If preferred device is set, the ID of the preferred device will be stored as the "preferred_device" context element. When no device is set as preferred device, it holds the value "undefined". Lastly, if there are no available devices currently connected to PMS, the value of this context element will be "no_connections".

A class PMSClientContext will be implemented to hold information about the context of a device. All of the client context objects will be stored in a map structure, which in turn is added as an element, "client_context", to the repository. The PMSClientContext objects store information about device location, bandwidth and the last time a heartbeat we received from the device.

The next sections will give an overview of the most important components that are required for performing session migration in PMS.

4.8.3 PMSClient component

The server will store information about each of the devices that are currently available. This information includes an unique ID for the device, ip-address, port. This could also be extended to include information about other of the device's properties, such as screen size, resolution etc, but this will, as mentioned in the above section about context, not be done for this PMS in this thesis.

Each of the clients available to the server will be represented as *service mirrors* in the QuA platform, and will be realized through a class called PMSClient. Service mirrors are QuA's way of representing each of the available implementations of a service type. Through planning, which includes error prediction and utility calculation, QuA automatically selects the service mirror which yields the expected best user experience. Planning and utility calculation will be discussed later in this chapter. The PMSClient class will contain a UDP sender component, which will be set up to be used to send UDP packages to the device the service mirror represents.

This UDP sender component will also be realized using QuA (service mirrors). This might not be an optimal solution performance wise, but makes for easily changing it to use a different protocol.

4.8.4 PMSListener component

To receive incoming messages from the clients, the PMS server will have a listener component which runs in a separate thread listening for incoming messages on a given port, using a UDP receiver component. It will then parse the incoming messages and handle each of the incoming requests. If a new client is connecting, the listener will be responsible for setting up the client information, reply to the client, and advertise the new information to QuA. The listener will also handle other incoming information from the clients, such as updated context, a device is set as *preferred device*, and disconnection.

Another task of the listener is to monitor the clients heartbeats. If a client fails to send a heartbeat over a given time period it will be removed as if it had disconnected from the server. For this part the weaknesses of UDP communication will have to be taken into account.

Depending on what sort of request is received from the client, a reconfiguration of the PMS server will be triggered by the *listener*. If a client connects and a device is already set as preferred device, a reconfiguration is not necessary. The same goes for disconnection if the *preferred device*Â is not the disconnected one. However, if a *preferred device* is not set, and which device to stream to is decided by QuA, calculating utility for each device a reconfiguration might be necessary and the adaptation manager is triggered.

The listener listens on the same port for messages from all the clients, which means that each incoming message needs to contain the client device's identification key. When an incoming message is received, the listener thread will parse it into id, command and arguments, and are then handled according to the given command.

Since QuA has no convenient way of recovering a specific service mirror that has been advertised, a workaround will be used, where the listener keeps a *hashmap* for storing references to the service mirrors. The client IDs are used for keys in the hashmap. This hashmap makes recovering a device's service mirror much easier for later requests from that device.

A summary of each command and how it is to be handled by the PMSListener component follows:

The "Con" command

A device sends a message with the command "con" when it is connecting to the service, that is letting the server know of its existence. The device also

includes a requested port for which it wants the video stream sent to. The port could be static, but allowing the user to select a port for each of the devices makes testing the system on one machine easier.

When a "con" request is received, a new *PMSClient* service mirror is instantiated and initiated for the requested port. A message including the clients devices ID is returned to the client before the new service mirror is advertised to the QuA platform. A reference to this service mirror will be stored in the listeners *clients* hashmap.

A PMSClientContext object will be created and initiated for the new client device and added to the *client_context* hashmap in the context repository. Client bandwidth and position information is piggybacked as arguments in the "con" message. The *preferred_device* is set to "undefined" if its present value is "no_connections". Finally the *client_trig* context is set to 1 to trigger the adaptation manager if *preferred_device* is "undefined" or "no_connections", that is, no device is currently set as preferred.

"dcon" command

The "dcon" command is sent by the device (client application) when it wishes to disconnect from the server. The service mirror for the client that sent the disconnect message will have to be removed, so that it is not considered when planning occurs the next time. The context information about that device will also have to be removed.

The service mirror for the device that sent the "dcon" message will acquired from the *clients* hashmap and will then removed from it. Since QuA does not yet support removing service mirrors that have been advertised from the QuA repository, a workaround will be used to prevent this service mirror to be considered by the planner. This is done by changing the service mirror's state to *ARCHITECTED_STATE*. This is not a nice solution, but will do the trick and can easily be changed if functionality for removing service mirrors is implemented in the future.

The client context object for this client will be removed from the *client_context* hashmap in the context repository. If the *preferred device* context is currently set to the disconnecting device it will be changed to "undefined". Another check on the size of the *clients* hashmap will tell if the disconnecting client device was the only one currently available to PMS. The *preferred device* context is then set to "no_connections". The adaptation manager will then be triggered. This will be done regardless of if the disconnecting device is the one that is streamed to, this because there is no good way of checking if a service mirror is the one that is currently active.

”hb” command

The client application sends a heart beat message at regular intervals. This message will have the ”hb” command, and will also piggyback the device's context information. The device context information in the message is used to update the senders *PMSClientContext* object in the context repository. This object will also store the current time whenever it is updated. This is done to allow the *PMSListener* to discover if a device is no longer available. If too long time has passed, the device will be ”disconnected” as if a disconnect message was received.

”pref” command

The ”pref” command is issued by a device only when the user explicitly requests it. This message only contains the command and the device id. Regardless of what the *preferred_ device* context variable is already set to, it is now set to the ID of the device that sent the message, so in a way it 'hijacks' the video stream. This is done to enhance the user experience. If a user moves from one device that is set as the preferred device to another, but forgets to remove its status as preferred, he will not have to go back to the old device to do this, but can simply make the new one the preferred device. This certainly also allows for some problems if two users try to control the same video streaming session at the same time, moving it back and forth.

”npref” command

The ”npref” command is sent by the client application if the user has chosen to free the stream, that is, removing the preferred device restriction. The *preferred_ device* context variable is set to ”undefined” to once again allow the QuA to select which device to stream to by calculating utility. This command can be issued by any of the available devices, not only the one that currently has status as preferred. However, the ID of the device will included in the message for making sure that the message is sent from a device that is currently registered with the PMS server.

Handling unexpected behavior

Unexpected behavior, such as receiving a command with an ID that is unknown, or receiving an unknown command, will simply print an error message and be ignored. For all incoming commands except the ”con” command

the ID given in the message will be checked against the clients hashmap to ensure that a PMSClient service mirror exists with that ID. The PMS server application will always continue running as normal.

4.8.5 Planning and adaptation

When planning, the utility function will calculate utility for each of the currently available devices to decide which one is to be selected for receiving the video stream. If a the user has set a device as the *preferred device* the utility function will filter each of the clients ID on the one of the *preferred device* only calculating utility for the devices that conform to this value. This utility value will be calculated based on each devices current context.

Filtering on the *preferred device* context before calculating the utility will make the process of adaptation and reconfiguring much faster, as the calculating of utility is a time and resource consuming process depending on the number of available clients. This will make session handover faster and smoother. If a user wants the video stream to a given device and sets it as the *preferred device*, the video will continue to stream to the device which is currently receiving the stream until reconfiguration is complete and is then continued at the new device instantly, but there will be a time delay from the user sets a device as preferred and the video stream actually being migrated to that device. Filtering on the *preferred device* context and minimizing the time used for reconfiguration will shorten this time delay giving a better user experience.

The adaptation manager will be extended to allow for triggering replanning. This is needed e.g. when a device connects and PMS needs to plan to see if this device gives a higher utility, as was explained in the last section. This is done through monitoring a context variable and performing replanning if it is set.

The adaptation manager is pull based, which means that it polls for change in context at regular intervals. If there is significant change, it will trigger a replanning of the application. This checking of context will be expanded to also check if there has been significant change in the locations of the user or the devices. If there has been significant changes, replanning is triggered. In order to do this, the adaptation manager will have to store the context of the last replanning. Then, for each polling interval these old context values will be compared to the current values.

4.8.6 Calculating utility

Utility will only be calculated if the *preferred device* context does not specify a device. Utility is calculated based on the context of each of the available devices and user context.

The error estimator function will first estimate the best possible video quality, given the current bandwidth, and also the distance between the device and the user. These values will be normalized to a value between 0 and 1. A possible video quality of 1 means that all the layers of the video can be sent. A value of 0 means that no video can be sent. The distance between user and device will also be normalized. All devices which are more than 100 units away from will be given a value of 0, which means that that device is too far away from the user and should not be considered. A device at the exact same location as the user is given a value 1. Utility for distance linear, and is negative proportional to distance from user. The new utility function for PMS can be represented as follows:

$$U(t, y, c, d, r, l) = W_t K_t(t) + W_y K_y(y) + W_c K_c(c) + W_{d,r} K_{d,r}(d, r) + W_l K_l(l)$$

This is essentially the same utility function as the old one, which can be found in section 2.2.8, except that a dimension (l) for distance between user and device has been added.

The values acquired from the error estimator are assigned weights. For PMS these will, as was mentioned in section 4.3 distance and quality of video will be assigned equal weight. Using these weights, the utility function will calculate a total utility value which is returned to the planner.

Chapter 5

Implementation

This chapter will cover specifics of the implementation not covered in the design chapter, as well as issues discovered during implementation.

5.1 Choosing technology

There was not much choice for technology for the implementation to be done in this thesis. The reason for this is that PMS is implemented using QuA and both PMS and QuA are implemented using Java. So, for extending the PMS application by implementing support for session migration, Java and QuA was used.

For the client application the QuA middleware technology was not used. The reasoning for this can be found in section 2.1. The client application could have been implemented using a different programming language, but using Java, a lot of the components used for implementing the client application in the original implementation could be reused. Because of this, the client application was implemented using Java.

5.2 Communication

As was described in the design chapter, communication between the device and the server was done using messages sent using UDP. The commands, that are sent from the device to the server, are implemented using simple ascii strings where the commands and the arguments are separated using the '|' sign. This makes testing and understanding the code easy.

5.3 Implementing the client application

5.3.1 Overview

The client device is fairly simple, it's only task being to communicate with the PMS server and receive and display the video stream. The different components for decoding and displaying video, as well as sending and receiving messages are implemented by simply reusing the components of the original PMS application (*DMJMultiThreadedVideoDecoder*, *DMJGraphicalVideoSink*, *NetUDPSender* and *NetUDPReceiver*), however in the client application they are not running on the QuA platform, but are implemented as regular Java objects.

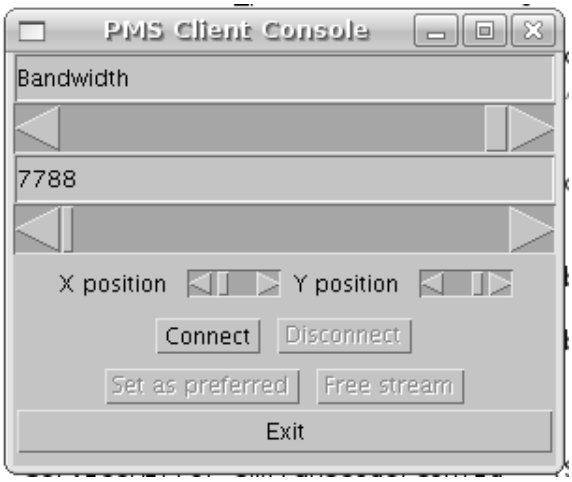

Figure 5.1: Client application GUI.

A simple GUI lets the user control the application. The GUI is shown in figure 5.1. Buttons let the user connect and disconnect. A button labeled "set as preferred" will tell the PMS server that the user wants this device set as preferred device, and thus wants the video stream sent to it. Another button labeled "free stream" will remove the device's status as preferred device, allowing the PMS server to decide which device to stream to based on utility calculations. Scrollbars allow the user to adjust the client's context variables; bandwidth and position (x and y coordinates). Bandwidth is displayed in a text field and can also be changes directly by typing in it. The second scrollbar lets the user select the port he wants to receive the video on. This was done to make testing the application while implementing simpler, as all the client applications were ran on the same computer.

The video decoder and the video sink will not need to be stopped or paused

when the application no longer receives any video from the server. The video decoder will simply decode any encoded video that arrives and forward it to the video sink which displays the video on screen. The components will not fail if they stop receiving video, or receives a corrupt video frame. By corrupt video frame is meant a frame that is not an intra frame when the decoder does not have the earlier video frame information to render the received frame correctly. An error message is printed but the components continue as normal. This makes session transfer easier since there is no need to consider smooth handover at the client side, but the client can continue as normally without being the currently selected client for receiving the video stream.

5.3.2 Session control specifics

Session control communication between the client and the server is done on two different ports. This is done to allow easier testing of the system with several clients on one computer. Messages from the client to the server is done on a fixed port, however messages from the server to the client application is set to be received at a port-number 100 higher than the port on which the user has requested to receive the video stream. This is done to prevent messages intended for different clients to be sent by the server on the same port. In a more finalized version of the PMS system the port numbers would likely be hard-coded. After all, the PMS system is a distributed system and there is no reason to running several instances of the client application on the same device.

5.3.3 Improvements

This section presents some possible improvements and additions to the implementation of the PMS client application.

It might be desirable to have a way for the client application to discover if the connection to the server is lost. This could for instance be done by having the server send ack messages when it receives a heartbeat from a device. It could also be desirable with more feedback from the server, allowing the device to know if it is supposed to be receiving video or not at any given time.

As all testing was done running both server and all client application instances on one computer, parsing of ip addresses has not been implemented. This obviously needs to be fixed for the client application to able to run on a different device than the server.

5.4 Implementing the server application

Most of the specifications for the implementation of the PMS server application has been explained in the design chapter. However, there are some details with the implementation which should be explained in detail. This section will present these, as well as issues discovered when working with the implementation.

Suggested improvements to the server application will be discussed in chapter 7 and 8.

5.4.1 Issues discovered

The PMS server application uses several threads for doing different work; e.g. PMSListener, PMSLocationDisplay, Adaptation Manager etc. Most of these threads need to access the context repository. To prevent these threads from accessing the data at the same time, preventing dirty reads, etc., synchronization had to be used. This might in some instances slow down the performance of the system, but it will ensure consistency.

An issue with QuA that was encountered when working with the implementation was the use of constructor functions in component classes. Constructors are not defined in interfaces, and thus QuA has no knowledge of these. This will make QuA crash without any good feedback as to what is the problem. For developers new to using QuA, this should be noted. The solution to this would be to create for instance an init function, specify it in the interface, and call it after creating an instance of the component. It is also difficult to debug applications implemented using QuA.

5.4.2 Context

The original PMS application had the option to set the available bandwidth between the device and the server using a scrollbar. This scrollbar is still in use, and can be seen as setting the servers outbound bandwidth. When estimating error for a device, the lowest value of this bandwidth and the device's bandwidth will be used.

User and client positions are restricted to and area of 100x100 units. As devices more than 100 units away from the user is regarded as useless, any devices outside this boundary would not be considered when planning. This area and these units are only defined for making testing easier.

5.4.3 PMSLocationDisplayImpl

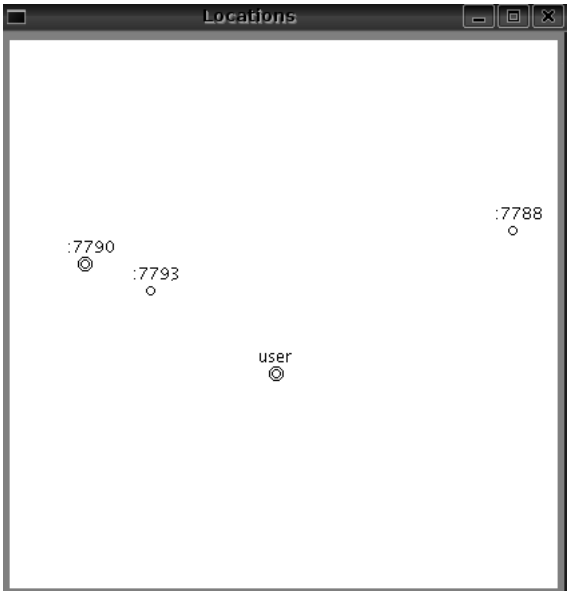

Figure 5.2: PMS location display

The PMSLocationDisplayImpl is a simple component for the PMS running a GUI thread to visualize user and device locations. It is realized using only simple Java AWT graphics. A black background serves as the environment. Devices are represented by yellow circles with the device id labeling them. The currently active device (the one receiving the video stream) is green and blinking. A blue blinking circle indicates the user.

The user and device positions is retrieved from the context repository. For drawing the devices, the hashmap of device context is iterated through and each device is drawn according to its position. All the position values are scaled so that they fit the size of the display. To draw the currently selected device a different colour, the *PMSLocationDisplay* has to have some way to knowing which is the currently selected service mirror. Due to some QuA limitations in getting the currently selected service mirror a workaround was used. By having QuA set a link to the *PMSClient* object for each reconfiguration (the same way it is done for the *PMSDummyTranscoder* component) it is possible to use this to get the ID of the active device. By comparing the ID of the device that is to be painted to this ID the active device can be drawn a different colour.

The colours of the devices are dependent upon their current available bandwidth. A device with "full" bandwidth is displayed as completely green, while

a device with no available bandwidth is red. Devices with bandwidth values in between are displayed with corresponding hues between green and red. Clicking the mouse inside the window will move the user to that location, updating the context repository with the change.

This GUI gives a nice overview of the position of the devices compared to the user and makes it easier to see how the utility function selects what service mirror to use based on the distance between the user and device.

Chapter 6

Testing

In this chapter the performance and correctness of the implemented solution for doing session migration in QuA using service mirrors are tested. The time used for performing session migration is an important factor for the user experience. This and correctness will be the focus of the tests performed.

6.1 Test environment

PMS is meant as a distributed system, but this has no impact on the service planning and session migration process performed by the PMS server as they are all performed without communicating with the client. Thus, due to the complexity of testing the application with the client applications running on different devices, as the system was meant to be used, the testing will be done by running the server and then have all service mirrors for client devices created directly by the testing component. As mentioned, this will not affect the test results concerning the performance of the session migration process internally in the PMS application.

The specification of the computer used for running the test is was as follows: Intel Pentium D 940 (2x3.2GHz) with 2GB memory. The operating system used was Ubuntu (Release 7.10 (gutsy)) running Linux kernel 2.6.22-14-generic. Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_13-b05) was used for executing the Java code.

6.2 Performing tests

For testing the implementation of session migration in PMS, a new component, responsible for running the tests of the system, was implemented. This component runs a separate thread which performs all of the different tests, writing the results to a local file. The tests measures the performance of the session migration functionality and time used for service planning.

The smooth handover part of session migration will not be tested, as this is very much subjective. Some users might rate this differently, and indeed, smooth handover might not have such a large impact on the user experience as it is nearly impossible for the user to move eyes from one device to another at the exact moment a session migration occurs.

Due to time limitations, testing session migration with moving devices will not be done. Also, the tests are constructed so that the live configuration of PMS is always selected. This should not affect the performance, as all possible configurations of service mirrors are considered in planning.

The natural thing to do for testing would be to compare the performance of the PMS application, e.g. time taken for session migration or scalability, to an already similar application or different implementation of the PMS application not running on QuA. However, there is no real candidate for such a comparison, so such testing will be left for future work.

As the focus of this thesis is implementing session migration on the PMS server, and the client application really has no other functionality than receiving and displaying the video stream, as well as simulating context, user interaction and each of the devices will be simulated. This will be done by setting up a *Sender* object for communicating with the *PMSListener* in much the same way as the client application does. Connect messages can be sent by this thread, making PMS generate and advertise mirrors for each of these. The fact that PMS uses UDP, and not some connection oriented protocol, for streaming the video makes it easier to test the application as no receiving end for the video stream needs to be set up. Giving the testing component, *PMSSMTester*, access to the context repository will allow it to manipulate user context variables directly removing the need for sending messages to the server for updating context, further simplifying the testing process. Having context manipulated directly instead of through sending messages will not have an impact on the test results, as what is being measured is the performance of the actual session migration process.

For measuring the time taken between changing a context variable, indirectly triggering application reconfiguration, and when the session migration has been performed and the right *PMSClient* service mirror has been selected,

the testing component will have a reference to the current *PMSClientImpl* object much in the same way as the *PMSDummyTranscoder* has. This will be set by QuA when a reconfiguration occurs. By finding the time it takes to change the reference from one device to another, it is possible to measure the time needed for session migration.

Although the actual session migration is delayed a bit by the transcoder to make handover smooth, this time delay will not be taken into account for the test results. The reason for this is that this delay would also occur in other applications doing session migration for video using a video codec based encoding frames by using differential coding. Including this delay in the test results would also induce a randomness to the results depending on what time a session migration occurs relative to the streaming of an intra frame.

The heartbeat functionality implemented to discover that the connection to a device is lost when using UDP will be disabled for testing. With this functionality on, the testing component would have to send a heartbeat for each of the connected clients at regular intervals. By disabling this, the implementation of the testing component will be simpler.

The following tests will be performed; a correctness test which assures that the correct device is selected, given the context; testing scalability of the session migration functionality when calculating utility for each client; testing scalability of the session migration functionality when setting a device as preferred device. Details of these test will be explained in the following sections.

6.3 Correctness test

In this test a scenario where user movement in an environment with four available devices are available is set up. The test is devised to check that PMS selects the correct device depending on the current context; user and device positions, as well as device bandwidth.

The testing component connects four clients and sets their positions as well as the user position. It then moves the user seven times, triggering replanning for each step. For each replanning it checks that the expected device is the one that is actually selected and write the results of the tests to the log file.

Table 6.1: Correctness test: device setup

Device (ID)	:7788	:7789	:7790	:7791
Position (x, y)	10, 60	30, 40	80, 60	80, 40
Bandwidth (kbps)	1000	1000	500	1000

Table 6.2: Correctness test: user positions

Position	1	2	3	4	5	6	7	8
User coordinate (x)	10	15	25	45	60	80	80	80
User coordinate (y)	50	50	50	50	50	50	53	58

6.3.1 Test setup

Table 6.1 shows the setup for each of the devices available in this test, and table 6.2 shows the initial position for the user, as well as the position for each of the movements made by the user in the test. Device IDs start at *:7788* and are incremented by 1 for each connected device. Figure 6.1 illustrates the path of the user. It should be noted that device *:7781* has much lower bandwidth than the other devices.

6.3.2 Expected results

Table 6.3: Correctness test: expected results

Position	1	2	3	4	5	6	7	8
Expected device (ID)	:7788	x	:7789	x	:7791	x	x	:7790

Table 6.3 shows the expected results for this test; which device should be selected if the planning and utility calculation in PMS performs as it is supposed to, for each of the users position. The 'x's indicate that no session migration should occur. These expected results are also illustrated in figure 6.1, where the 'm' indicates a point where session migration occurs and 'x' indicates that no session migration should occur.

The expected results are for PMS to always select the device that is closest to the user, except in position 7 where device *:7791* should be selected even though *:7790* is closer. This is because *:7790* has a much lower bandwidth than *:7791* which means that it can not receive as high quality video. In the last position, however, the user is so close to *:7790* that it will be selected regardless of the low bandwidth. This is due to the weights assigned to the QoS dimensions distance and video quality.

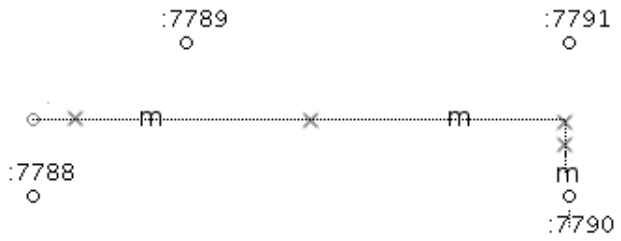

Figure 6.1: Correctness test - user path.

6.3.3 Actual results

For this test PMS performed as expected and selected the correct device each time the test was run. This is a good indication that the planning and utility calculation mechanisms of PMS are performing correctly.

6.4 Replanning scalability test

This test measures the time taken for session migration, from context change to the new client device has been selected, for different numbers of available client devices. The test performs session migration for 5, 10, 25 and 50 clients. Triggering service planning is done by setting a context variable *test_ trig*, and the adaptation manager has been expanded to trigger replanning of the application if this is set.

This test gives a good indication of how well the session migration for PMS using service mirrors in QuA works for an increasing number of users.

6.4.1 Expected results

The time taken for session migration is expected to grow linearly as the number of available clients does. For each additional available client the utility calculations will have to be done. The utility calculation is the same for all client service mirrors, thus the time taken for utility replanning is expected to grow linearly with the number of available clients. For each available device 64 different configurations of service mirrors (16 for live +

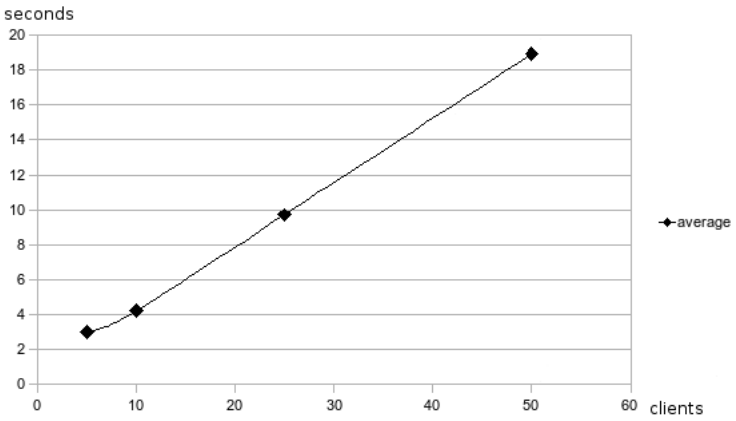

Figure 6.2: Average time for session migration.

48 for time-shift) are considered by the planner. This means that for 50 available devices, 3200 configurations, plus 1 for the storage configuration, are considered by the planner.

6.4.2 Actual results

Table 6.4: Replanning scalability test - result samples and average

# clients	sample1	sample2	sample3	sample4	sample5	sample6	average
5	3.1s	3.2s	2.9s	2.9s	3.0s	3.0s	3.0s
10	4.5s	4.5s	4.2s	4.1s	4.1s	4.4s	4.3s
25	10.2s	11.4s	8.9s	9.4s	9.3s	9.1s	9.6s
50	22.2s	18.5s	17.2s	19.0s	18.9s	17.6s	18.7s

Table 6.4 shows some sample results from the tests done, as well as an average time for session migration for each number of available clients. Figure 6.2 shows a chart which plots the average time, from this we can see that the expected result of the test seems to be approximately correct. Time taken for replanning and session migration correlates to the number of available users. This implementation with session migration performs extremely slow compared to the PMS implementation without session migration. This is probably because of the high number of configurations of service mirrors that are considered by the planner; 3201 vs. 64 configurations.

6.5 Preferred device scalability test

This test is very similar to the replanning scalability test, but instead of triggering the adaptation manager by setting the *test_ trig* context variable forcing replanning, the testing component sends a message to PMS using UDP, setting the last connected device as preferred device. This is done for the same numbers of available users as in the replanning scalability test. When testing for each of the number of available clients is finished, the *preferred_ device* context variable is set to *undefined* to clear it for further testing.

The resulting times of this test will be compared to those of the previous scalability test.

6.5.1 Expected results

This test is expected to perform session migration much faster than the scalability test. The reason for this is that the utility function will filter on the *preferred device* context. This means that utility will not be calculated for devices which has an ID that does not match the ID of the preferred device.

6.5.2 Actual results

Table 6.5: Preferred device scalability test - result samples and average

# clients	sample1	sample2	sample3	sample4	sample5	sample6	average
5	2.8s	2.9s	2.6s	2.8s	2.8s	2.8s	2.8s
10	4.1s	4.5s	4.4s	4.1s	4.2s	3.9s	4.2s
25	10.1s	9.5s	9.0s	9.3s	9.4s	8.9s	9.4s
50	19.8s	18.2s	17.0s	19.1s	17.8s	18.8s	18.5s

Table 6.5 shows some sample results from the tests done, as well as an average time for session migration, for each number of available clients, when setting a client as preferred device. Figure 6.3 shows a chart which plots the average time of the test performing utility calculations against this test which does not. This chart clearly shows that assuming that the calculation of utility would count for a big part of the time taken for session migration was wrong. The preferred device test performs only slightly faster than the other test, and the difference would be virtually unnoticeable to the user.

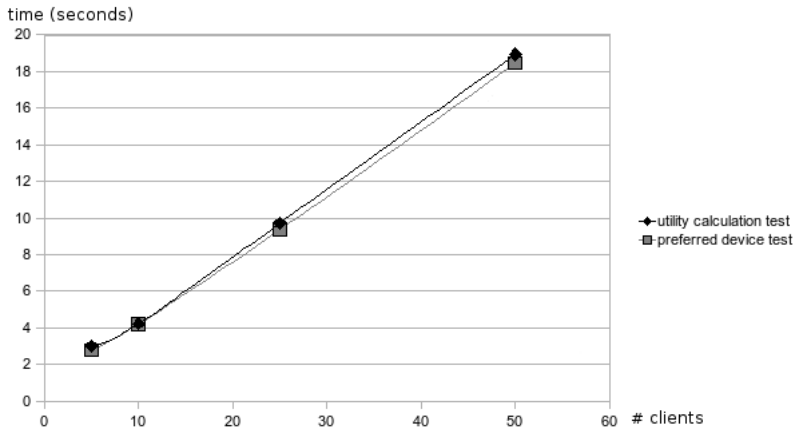

Figure 6.3: Average time for session migration with preferred device

Chapter 7

Evaluation

In this chapter, the performance of the session migration functionality implemented for PMS will be analyzed and evaluated based on the results of the tests in chapter 6. This performance will be compared to the requirement specifications specified in chapter 3.

7.1 Correctness and consistency

The test for correctness in planning for session migration provided the expected results, which shows that using planning based middleware for performing session migration is possible. Using both position of user and of devices as context in this planning phase works well, but the weight ratio needs to be adjusted through more thorough testing and evaluation. These weights could also be left as user preferences, but there should probably be provide some default configurations for the user to chose from as this might be too complex to non-expert users.

7.2 Scalability and performance

Both of the tests done to check scalability of PMS, when it comes to number of users, performed without faults. This means that the implementation of PMS is able to handle the number of users specified as reasonable in the requirement specification without making wrong decisions or crashing. The performance results of planning in the tests with many users, though, were much higher than desirable and expected. The test where scalability of planning for session migration when preferred device was set provided

unexpected results as the time taken for planning was approximately as long as for planning when preferred device was not set. This section will analyze these results and try to provide some insight into the reasons for these unexpected results.

7.2.1 Planning calculating utility

In the test *replanning-scalability-test* the performance of the session migration functionality with calculating utility for all available devices was tested. This test was run for 5, 10, 25 and 50 available devices (PMSClient service mirrors). The results of the test shows that the time taken for planning grows linearly with the number of available devices. There is a certain constant of time taken for planning + time for calculating utility for each device.

The total time taken for session migration in this implementation is much higher than would be desirable and expected. For few devices (5-10) the replanning takes up to 5 seconds. This means that from context change is discovered and adaptation and replanning is triggered to actual session migration has occurred (not taking into account waiting for an intra frame for smooth handover) takes up to 5 seconds. This is an acceptable delay since a user would not expect to see instant session migration the time he for instance walks into one room too another. It would take some time for the user to change his view from one device to another.

However, as mentioned, the time taken for planning grows linearly with the number of clients. This is a much higher rate of growth than is desirable. For 50 available clients this means that delay from context is discovered to actual session migration occurs reaches as high as more than 20 seconds. This is a significant delay and would greatly hamper the user experience. It is possible that a user would lose over 20 seconds of the video he was watching maybe missing out on an important event in the news broadcast or movie he was watching. As the video continues on the previous device until session migration occurs, the whole video would be displayed, but it might be displayed on a device that is outside the users viewing range. For instance if the user walks from one room to another, his change of location triggering session migration, the video would still be displayed on the device in the other room until it is migrated to the one in the new room. However, as the previous device is in another room, the user cannot see it.

The obvious solution to this problem of loosing video due to too high delay would be to find a way to make planning of the application faster. There are several ways that the planning phase could be improved to perform faster. When calculating utility and estimating error for each device there are several

calculations that are performed many times. If one could find a way of calculating each only one time, and remembering the results for each time they are calculated, one could greatly reduce the time taken for planning.

Another solution which might be easier to implement, but not as optimal, would be to create a new configuration for PMS when session migration occurs. This configuration would start buffering video at the time that a context change is discovered and then starting to stream the buffered video when planning for session migration is finished. If this solution was to be implemented one should prevent planning from happening too often, or else the buffer might grow bigger and bigger. One could also use time-shift to stream at a faster rate to empty the buffer after a session migration, else the buffer would grow for each session migration. If PMS handles a continuous stream, for instance a television broadcast, the buffer might overflow if PMS runs for a long time and many session migrations occurs. One major issue that needs to be resolved for this solution is when there is planning, but no session migration is to occur and the planner selects the same device as the one that was already selected. In this situation we would not want PMS to buffer, because no session migration is occurring. This could for instance be done by streaming the video to the old device while buffering and then dumping the buffer if the same device is selected.

7.2.2 Preferred device planning

The *preferred-device-scalability-test* provided the most surprising results of the tests. The test was set up exactly the same as the *replanning-scalability-test*, but instead of calculating utility for all available devices, preferred device and filtering was used. This test was expected to give much lower times for session migration than the other test, but as the results show, the time taken was almost as high as for the first test and the difference would be as good as indistinguishable to the user.

Optimal time taken for session migration for preferred device would be the same time as taken for calculating utility with only one available device regardless of how many devices are available. This is because when setting preferred device PMS already knows before planning is done what device should be selected. However, this PMS solution was not expected to perform this fast, as it uses QuA and planning to select the device through filtering on ID on the preferred device's ID. As in the other test this test was expected to provide times for session migration growing linearly with the number of users. This proved to be correct, as can clearly be seen section 6.5.2. However, the times in this test were expected to be drastically lower than for the test which calculates utility for each device. The filtering is implemented such

that the first thing done in the error estimator function is filtering, checking ID of the device (service mirror) currently being planned for against the preferred device ID and raising an exception if they are not equal. This fact shows how little of the time used for planning and session migration is actually used by calculating utility. The results for 50 available devices shows that preferred device performs only about half a second faster than when calculating utility. Of the total time for planning for the application with 50 available devices, only approximately 2.5% is used for calculating utility. This indicates that most of the time used for replanning is internally in QuA, and caused by the number of possible configurations of service mirrors; 64 different configurations for each device + 1; 3201 configurations for 50 available devices.

If PMS was not implemented using planning based middleware time taken for selecting device when knowing which device is preferred would be much faster. However, there are some ways that the PMS solution using QuA could be made to run faster. These solutions involve possible improvements to the QuA architecture and implementation. One functionality that would be desirable in this case would be the ability to filter service mirrors earlier than in the error estimator. As it is now error estimation is triggered for each of the possible configurations of service mirrors. If it was possible to set some constraints on service mirrors, for instance that they are only usable given certain context, earlier filtering could be applied so that only configurations with service mirrors which fulfill the criteria of the constraints are used for error estimation and utility calculation. This would greatly improve the time taken for planning in this case. If PMS was implemented in such a way that more than one service mirror for device was used in each configuration growth would be exponential and this functionality would be even more important.

7.3 Smooth handover

Smooth handover was not tested in the tests in chapter 6. The main reason for this is that smooth handover is difficult to measure, thus also difficult to test. A workaround was used to make session migration in PMS perform smoothly. By workaround is meant that QuA was not used to control smooth handover, but reconfiguration of the *PMSClient* service mirror (representing the device) was delayed in the transcoder by not changing it before an intra frame was about to be sent. This provides some semblance of smooth handover as the displaying of video is transferred from one device to another instantly without any loss or corruption of data.

However, in light of the results of the tests done with respect on performance

and scalability, a different sort of smooth handover might be desirable. As mentioned in section 7.2.1 there might, for high numbers of clients, happen that session migration and planning takes a lot of time after the user has moved, and some of the video might be lost to him. In this situation, in a sense, smooth handover is not really achieved. As described a solution where buffering is done for every session migration is a possible solution to this problem and would give the user a better sense of smooth handover of the video stream.

As this other solution to smooth handover might be very difficult to realize, and that users seldom has more than 5-6 devices capable of video streaming, the current solution is performing adequately for the needs of PMS. If the solution of buffering was implemented, this would lead to even more complex mechanisms in QuA making time delay for session migration even higher. This in turn might not give as good a user experience, at least when there is not a very high number of available devices.

7.4 Issues with tests

QuA has no way of measuring the time taken for planning in a good way. This means that the tests had to be implemented in such a way that a loop was running and waiting for the new device to be selected. This loop sleeps for 100ms each time to prevent it from taking up too much processing power. Due to the way the tests were implemented the timing results given in the test are not completely accurate. Because the loop checking to see if a session migration has occurred sleeps for 100ms the tests are only accurate to one tenth of a second. These small inaccuracies have no great impact on the results, but should be noted. It might be desirable for application developers to have a proper functionality in QuA for measuring the performance of planning. This would make optimizing the code easier.

7.5 Summary

The tests done show that the implementation of session migration in PMS using QuA and service mirrors for automatic adaptation and session migration is possible. The solution performs tolerably well for low numbers of clients but has significant delay when the number of available clients is high. There are several things that could be optimized to make the solution better.

Chapter 8

Conclusion & Further Work

This chapter will conclude the research of this master thesis and suggest further work, as well as give an insight into the experience of working with the QuA planning based middleware. The first section will conclude the findings of the research done, then follows a section where experiences gained from this research are presented and discussed. The last section presents suggested work for the PMS application as well as suggested changes and additions to the QuA middleware.

8.1 Conclusion

The motivation for the research done in this master thesis was the desire to be able to move a video streaming session from one device to another. This was proposed done using planning-based component architecture middleware to perform automatic context sensing and adaptation of the streaming session to provide a seamless user experience when moving between devices.

As has been shown through the implementation, and the testing and evaluation, this is a possible solution to the problem of session migration for a video streaming system. The evaluation of the results from the testing suggests that a lot of optimization of the implementation would be desirable. For a low number of available devices the implementation performs sufficiently well, but the results could and should be much better. For a high number of devices adaptation and planning, thus session migration, takes much too long time and needs improvement.

Using user position compared to the device positions, as well as device bandwidth, for planning which device is to be selected is easily done using QuA

and using these context elements brings the session migration solution closer to the future of ubiquitous computing. This would prove very useful if one for instance has one television screen in each room.

8.2 Experiences working with QuA

There are several advantages and disadvantages to using QuA. This section will outline some of the experiences gained when working with the QuA middleware.

To be able to successfully implement an application or a component using QuA extensive knowledge of how QuA works and its architecture is needed. Much study work is necessary to gain a satisfactory understanding of the middleware before starting to use it for implementing an application. Once this is gained QuA provides a good way of handling adaptation and evolution of context aware applications. It makes changing and adding to services easy and handles adaptation well, making much of the underlying decisions transparent to the developer.

The biggest disadvantage of QuA is that it is hard to debug applications while developing. This is because the architecture of the application becomes more dynamic than in regular object oriented programming and one does not have access to all the same information. This makes the approach to debugging the application very different and makes the development of the application more difficult.

8.3 Further work

8.3.1 PMS

There are several things that could be done for session migration in PMS to perform better and give a better user experience. There are also a lot of parts that are necessary for a finished product that has not yet been implemented. This section will suggest and outline some of these improvements and further research areas.

Context sensing components should be designed and implemented. For the implementation of PMS in this thesis all context sensing was simulated by simply inserting values directly or through user input (like setting bandwidth with the scrollbar in the client application).

The context used for planning for session migration could also be extended to also consider things like screen size and resolution, privacy etc... This might give a better user experience by more often correctly selecting the device that the user would prefer. This could also be done by having the application and planning learn from earlier selections, using this information in planning. For instance if the user very often selects a device it might be natural for the planner to favor this device. By learning from previous choices the planning would become more accurate in predicting which device would give the best user experience.

This implementation simply uses the user's and devices' physical coordinates as location when calculating utility. Making the concept of location more abstract might improve performance as well as the user experience. For instance, instead of using physical coordinates, one could define areas like rooms. This way, when planning, all devices which are in a different room than the user could be filtered out. With the current implementation a device in the next room could be selected rather than one in the same room as the user if its physical location is closer. This problem would be solved by defining locations in this way and planning would take shorter time due to filtering. However, the gain in performance might not be that formidable as the tests in chapter 6 shows that not much time is saved by filtering the way it is currently being done. One of the down sides of this approach is that it needs setup before it can work and it needs a way to determine what area/room the user is in. A hybrid to these two approaches could also be considered.

The implementation of the adaptation manager in PMS is pull-based and uses polling for checking for context changes. It sleeps for some time, then checks to see if there has been significant change to any context variables, and triggers planning. Depending on the length of time the adaptation manager sleeps, this pull based implementation might waste resources, and reconfiguration might occur later than desirable. A solution to this might be to implement a push-based adaptation manager instead. This push based adaptation manager would be triggered only when context change happens and would reduce the latency caused by the polling interval in the pull based approach.

As video most often is accompanied by audio, indeed many videos would be virtually useless if sound was not included, the possibility to split a session, which contains both video and audio, into two separate sessions for each would be a highly desired functionality. This would allow for for instance showing the video on a television screen whilst playing the audio through the stereo. Session migration of audio sessions has many different aspects from those of video session migration which should be considered. For audio

it could for instance be a possibility to adjust the volume according to the distance from the output device to the user.

Duplication of a video session into two identical sessions displaying the same video on both of these devices at the same time might also be desirable. This poses many problems which should be solved, like increase in outbound bandwidth from the server.

Signaling and communication between client and server, and even in between clients, should be considered. Only a rudimentary solution has been implemented in this thesis. An approach using SIP for session migration was presented in section 2.3.8.

8.3.2 Suggested changes/additions to QuA

Through the design and implementation of session migration in PMS, using QuA and service mirrors, some desirable changes and additions to the current QuA architecture and implementation have been discovered. These would be beneficial when developing session migration for streaming applications. This section will outline some of these.

The existing QuA implementation lacks certain functionality for implementing smooth handover for session migration applications. The application developer has no control of when a reconfiguration is going to happen. If there was support for flagging when a reconfiguration is possible, smooth handover would be easier, especially whens streaming video which is differentially coded. Making sure that the application is in a safe state before reconfiguration would ensure that data is not corrupted or lost.

As was seen in the tests, filtering of service mirrors in the error predictor does not really save much time when planning. The possibility of filtering at an earlier stage in the planning, preventing much of the calculation done, would greatly improve the performance of planning where filtering is used. This could be done through constraints on service mirrors so that they are only usable when certain context criteria are met. Only service mirrors which meet these restrictions would be used for planning. This would be especially useful if we had a setup where more than one client service mirror was in each configuration (for instance with session splitting) to prevent exponential growth of time taken for planning.

In the PMS case, the user is allowed to explicitly specify the preferred device for receiving the video stream. An even faster way to handle planning in this case than even early filtering would be the ability to select one service mirror explicitly. In effect, all other devices are then excluded from the planning

process.

Measuring (elapsed) time in the experiments was not straight forward. Internal support for timing the planning phase in QuA would be desirable and would make testing and optimization of the code easier for the application developer.

Appendix A

Source Code

The source code for QuA and PMS can be found on a separate CD. This section explain where the source files for the implementation work in this thesis can be found, as well as provide instructions on how to compile and run the code.

A.1 QuA and the PMS Server Application

QuA - */PMS-SM/server/*
PMS application - */PMS-SM/server/src/apps/PMSApp.java*
PMS component files - */PMS-SM/server/src/comp/qua/pms/*
PMS interfaces - */PMS-SM/server/src/types/qua/types/pms/*

To compile the source code for QuA and PMS Ant is required. Access /PMS-SM/server/ and do:
ant -f build.xml
ant all

To run the PMS server do:
sh run-video-example.sh video2.yuv

A.2 The PMS Client application

The source code for the PMS client application can be found here:
/PMS-SM/client/

The PMSClient.java is compiled as a regular Java application. Some JAR files are needed to compile the application. The files needed can be found here: */PMS-SM/server/*

The following JAR files are needed:
-DMJGraphicalVideoSink.jar
-Netpipe.jar
-NetUDPReceiver.jar
-UDPMSubscriber.jar
-DMJMultiThreadedVideoDecoder.jar
-NetUDPSender.jar

The required interfaces are located in:
/PMS-SM/client/qua/types/

Bibliography

[1] www.tversity.com.

[2] Viktor S. Wold Eide, Frank Eliassen, Jørgen Andreas Michaelsen, and Frank Jensen. Fine granularity adaptive multi-receiver video streaming. *Fourteenth annual Conference on Multimedia Computing and Networking (MMCN07)*, 2007.

[3] Frank Eliassen, Eli Gjørven, Viktor S. Wold Eide, and Jørgen Andreas Michaelsen. Evolving self-adaptive services using planning-based reflective middleware. *ARM'06, November 27-December 1, 2006 Melbourne, Australia*, 2006.

[4] I. Burnett et al. Mpeg-21: goals and achievements. *IEEE Multimedia*, 10:60–70, Oct.-Dec. 2003.

[5] J. Rosenberg et al. Sip: Session initiation protocol. *IETF RFC 3261*, 2002.

[6] Komiya D. et al. Use cases for session mobility. *IETF draft-komiya-mmusic-session-mobility-usecases-00.txt*, 2006.

[7] Eli Gjørven. Qua middleware concepts and terms (as understood by the author today). *Unpublished manuscript*, 2007.

[8] Teodora Guenkova-Luy, Holger Schmidt, Franz J. Hauck, and Andreas Kassler. Service mobility with sip, sdp and mpeg-21. *9th International Conference on Telecommunications - ConTEL*, 2007.

[9] M. Handley, V. Jacobsen, and C. Perkins. Sdp: Session description protocol. *IETF RFC 4566*, 2006.

[10] Cristian Hesselman, Henk Eertink, Ing Widya, and Erik Huizer. Delivering live multimedia streams to mobile hosts in a wireless internet with multiple content aggregators. *Mobile Networks and Applications 10*, 2005.

[11] W. Li. Overview of fine granularity scalability in mpeg-4 video standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 11:301–317, March 2001.

[12] Tinghuai Ma, Yonk-Deak Kim, Qiang Ma, Meili Tang, and Weican Zhou. Context-aware implementation based on cbr for smart home. *IEEE International Conference on Wireless And Mobile Computing, Networking And Communications (WiMob'2005)*, 2005.

[13] Sujeet Mate, Umesh Chandra, and Igor D.D. Curcio. Movable-multimedia: Session mobility in ubiquitous computing ecosystem. *Proceedings of the 5th international conference on Mobile and ubiquitous multimedia MUM '06,December*, 2006.

[14] Johannes Oudenstad. The design and evaluation of a qua implementation broker based on peer-to-peer technology. Master's thesis, University of Oslo, 2007.

[15] Vincent Ricqueborg, David Menga, David Durand, Bruno Marhic, Laurent Delahoche, and Christophe Loge. The smart home concept: our immediate future. *Industrial Electronics, 2006 1ST IEEE International Conference on*, pages 23–28, 2006.

[16] Sumit Roy, Bo Shen, and Vijay Sundaram. Application level hand-off support for mobile media transcoding sessions. *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video NOSSDAV '02*, 2002.

[17] Henning Schulzrinne and Elin Wedlund. Application-layer mobility using sip. *ACM SIGMOBILE Mobile Computing and Communications Review, Volume 4, Number 3*, pages 47–57, 2000.